EPFL

Thèse n° 9768

Just-in-time performance without warm-up

Présentée le 28 février 2020

à la Faculté informatique et communications
Laboratoire de méthodes de programmation 1
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Denys SHABALIN

Acceptée sur proposition du jury

Prof. J. R. Larus, président du jury
Prof. M. Odersky, directeur de thèse
Prof. V. Adve, rapporteur
Dr A. Prokopec, rapporteur
Prof. E. Bugnion, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

Acknowledgements

I would like to thank my advisor Martin Odersky, for providing me this unique opportunity to explore the realm of ahead-of-time compilation for Scala.

I am grateful to my thesis jury members James Larus, Edouard Bugnion, Vikram Adve and Alexander Prokopec for their time contribution to this thesis.

The work of Sébastien Doeraene on the Scala.js [35] has been one of the major inspirations for the project. A large number of the initial design and implementation in Scala Native is directly built upon the foundation that was laid by the Scala.js project.

A special thanks to Lukas Kellenberger and Valdis Adamsons whose excellent work on the runtime support for the garbage collection is one of the key factors in our final runtime performance results.

Colleagues at LAMP and Scala Center that I get to work with over the years including Martin Duhem, Guillaume Masse, Felix Mulder, Allan Renucci, Jorge Vicente and Ólafur Páll Geirsson.

Scala Native would not be possible without an incredible number of open-source contributions from the community. Including but not limited to Adam Singer, Adam Voss, Alex Dupre, Alexey Kutepov, Andrea Peruffo, Andrei Pozolotin, Andrew Smith, Andrzej Sołtysik, Ankit Soni, Brad Rathke, Carlos Quiroz, Christian Krause, Corey O'Connor, Cédric Viaccoz, David Patrick, Dubray Alexandre, Eric K Richardson, Ethan Atkins, Felix Garcia Borrego, Florian Duraffourg, Francois Bertrand, Greg Dorrell, Greg Oledzki, Gregor Ihmor, Hanns Holger Rutz, Henning Wielenberg, Henry Mai, Hubert Plociniczak, Jason Longshore, Jocelyn Boullier, Jonas Fonseca, Joseph Price, Kamil Tomala, Kenji Yoshida, Koji Agawa, Kota Mizushima, Lee Tibbert, Liudmila Kornilova, Marius B. Kotsbak, Martin Mauch, Mike Samsonov, Nadav Samet, Naohisa Murakami, Pablo Guerrero Rosel, Pawel Batko, Paweł Cejrowski, Paweł Krupa, Piotr Kwiecinski, Remi Coudert, Richard Whaling, Ruben Berenguel, Saleem Ansari, Sam Halliday, Sandeep Singh, Shadaj Laddad, Shunsuke Otani, Srepfler Srdan, Stefan Ollinger, Tim Nieradzik, Vincent Munier, Xavier Fernández Salas and Łukasz Indykiewicz.

Last but not least, I would like to thank Aggelos Biboudis for his time and patience reading the early drafts of my papers and of this thesis.

Lausanne, November 24, 2019 Denys Shabalin

Abstract

Scala has been developed as a language that deeply integrates with the Java ecosystem. It offers seamless interoperability with existing Java libraries. Since the Scala compiler targets Java bytecode, Scala programs have access to high-performance runtimes including the HotSpot virtual machine.

HotSpot provides impressive performance results achieved via just-in-time compilation. It starts program execution in interpreter mode, collecting profile feedback about called methods. This information allows HotSpot to identify hot spots in the program, which are then compiled on the fly to native code. This compilation scheme enables high peak performance at the cost of warmup time required to collect the profile data and perform just-in-time compilation.

This is a good example of the traditional tradeoff between ahead-of-time (AOT) and just-in-time (JIT) compilation. With AOT, compilers have less information, but the runtime story is reasonably straightforward. With JIT, compilers have more information, which enables advanced optimizations, but the runtime story becomes complicated.

In this dissertation, we present the design and implementation of Scala Native, an optimizing compiler for Scala. With Scala Native, Scala programs are compiled ahead of time, which avoids runtime compilation and enables instant startup times. On the other hand, Scala Native is able to match and supersede the peak performance of HotSpot on our benchmarks. In addition to that, Scala Native is a general-purpose Scala compiler - programs compiled by Scala Native closely match the behavior of programs compiled by the Scala compiler.

First, we introduce NIR, an intermediate representation designed with ahead-of-time compilation in mind. NIR represents programs in the single-static assignment form and has support for object-oriented features such as virtual dispatch and multiple inheritance. This representation is a key enabler of our compilation and optimization pipeline.

Secondly, we present Interflow, a link-time optimizer that takes advantage of the closed-world assumption to optimize the whole program at once. Our optimizer employs a number of techniques including partial evaluation, allocation sinking, and method duplication. The combination of these techniques allows Scala Native to outperform HotSpot on the majority of our benchmarks.

Finally, we describe how to improve runtime performance even further based on profile feedback. We propose a technique that splits methods apart isolating key hot paths that are then optimized more aggressively than the cold parts of the program. This provides a further performance advantage over HotSpot.

Keywords: Programming Languages, Compilers, Ahead-of-time Compilation, Link-time Optimization.

Résumé

Le langage Scala fut développé pour s'intégrer profondément à l'écosystème de Java. Il offre une interopérabilité fluide avec les bibliothèques Java. Puisque le compilateur Scala vise le bytecode Java, les programmes Scala ont accès à des environnements d'exécution haute performance, tels que la machine virtuelle HotSpot.

HotSpot produit des résultats impressionnant en terme de performances, obtenus grâce à la compilation à la volée (en anglais *just-in-time* ou JIT). Elle démarre l'exécution des programmes en mode interpréteur, tout en collectant des informations de profilage sur les méthodes appelées. Ces informations permettent à HotSpot d'identifier les « points chauds »(*hot spots*) du programme, lesquels sont alors compilés à la volée vers du code natif. Cette stratégie de compilation permet d'atteindre d'excellentes performances de croisière, au prix du temps de préchauffage (*warmup*) requis pour collecter les données de profilage et pour effectuer la compilation à la volée.

Ce qui précède est un bon exemple des compromis classiques entre compilation anticipée (en anglais *ahead-of-time* ou AOT) et compilation à la volée. Avec la compilation anticipée, les compilateurs possèdent moins d'informations, mais le déroulement à l'exécution est simple. Avec la compilation à la volée, plus d'informations sont disponibles – ce qui permet des optimisations avancées – mais l'exécution est plus complexe.

Dans cette thèse, nous présentons le design et l'implémentation de Scala Native, un compilateur optimisant pour Scala. Avec Scala Native, les programmes Scala sont compilé de manière anticipée, ce qui évite la compilation à l'exécution et permet un démarrage instantané. Scala Native est néanmoins capable d'égaler, voire surpasser, les performances de HotSpot sur nos benchmarks. En outre, Scala Native est un compilateur Scala d'usage général : le comportement des programmes compilés par Scala Native est très similaire à celui de ceux compilés par le compilateur Scala original.

Premièrement, nous introduisons la NIR (*Native Intermediate Representation*), une représentation intermédiaire conçue pour la compilation anticipée. La NIR représente les programmes sous forme statique à affectation unique (*static single-assignment form*, ou SSA), et présente des fonctionnalités dédiées à l'orienté objet telles que liaison virtuelle et héritage multiple. Cette représentation est un acteur clef de notre pipeline de compilation.

Deuxièmement, nous présentons Interflow, un optimiseur à la liaison qui tire parti de l'hypothèse d'un monde fermé pour optimiser chaque programme en entier. Notre optimiseur fait usage de plusieurs techniques telles que l'évaluation partielle, l'enfouissement d'allocation et la duplication de méthode. La combinaison de ces techniques permet à Scala Native de surpasser HotSpot sur la majeure partie de nos benchmarks.

Enfin, nous exposons comment davantage améliorer les performances sur la base d'informations de profilage. Nous proposons une technique dans laquelle les méthodes sont décomposées pour en isoler les chemins critiques, lesquels sont ensuite optimisé de manière plus agressive que les portions froides du programme. Cette technique produit un avantage supplémentaire en performances par rapport à HotSpot.

Keywords : Langages de Programmation, Compilateurs, Compilation Anticipée, Optimisation à la Liaison.

Contents

List of Figures

1 Introduction

Programming language design and implementation is a mature field of computer science that has been around for a long time. Throughout its history, it witnessed thousands of programming languages implemented in a myriad of ways. It is hard not to look at current results as mere reflections on the early works done by the pioneers of this field.

High-performance language implementations are represented by either *ahead-of-time* (AOT) or *just-in-time* (JIT) compilation. AOT compilation focuses on statically generating native code before the application is run, while JIT techniques may dynamically generate new optimized code at runtime.

AOT compilation has been traditionally the approach used for the *systems* programming languages that aim to expose the direct access to underlying hardware in the most direct way. The approach naturally fits into the overall picture of the lower-level programming and manual memory management as explored in languages such as C and C++.

On the other side, higher-level languages that tend to rely on some form of automatic memory management have been often implemented as interpreters. The need for higher-performance implementation of the interpreted runtimes gave birth to just-in-time compilation that bridged the performance gap through dynamic code generation.

Scala programming language was originally a part of the second camp of the just-in-time compiled languages. Its reference implementation targets JVM bytecode and relies on a deep integration with the Java ecosystem.

Research on the JVM runtime implementation has been traditionally dominated by the just in time compilers [9, 49, 71, 86] that emphasize the peak performance over the start-up time. The techniques used in JVM runtimes rely on multi-tier compilation model that starts with a baseline interpreter (or a simple compiler) that collects profile feedback that is used to selectively generate highly optimized code for the hot paths in the program [11].

While such design of the runtime implementation is able to deliver impressive performance

results, it has a significant overhead for short-running applications such as command-line tools.

As an alternative to this approach, in this thesis, we focus on purely ahead-of-time compiled compiler and runtime designed for the Scala programming language. While this allows us to avoid the JIT warm-up overhead completely, AOT compilation poses a major challenge to get comparable peak performance.

The main reason for the difficulty of obtaining comparable results lies in the fact that traditional AOT compilation does not allow any form of runtime code generation. All of the optimization decisions have to be done at compile-time without the ability to revert them later at runtime.

JIT-compiled runtimes, on the other hand, may compile the code based on speculative assumptions and deoptimize back to the baseline implementation if the assumptions under which the code was compiled are violated. This allows the runtime to dynamically adapt to the changes in the application behavior.

1.1 Design Goals

We present Scala Native - an ahead-of-time optimizing compiler for Scala that aims to fulfill the following design goals:

1. *Start-up time*. Language implementation should prioritize application start-up time as one of the key performance metrics. It must offer a significant reduction of the cold start overhead compared to the reference implementation.

2. *Peak performance*. The compiled program must run *at least as fast* as the same program running on top of the warmed-up reference implementation based on the JVM.

3. *Compatibility*. Our implementation should provide the same runtime semantics as the reference implementation. The error conditions should fail in the same way as the reference implementation, even if they are underspecified by the language itself. This is crucial to maintain portability of the existing applications which often depend on the implementation-defined behavior.

The existing compilers in the context of the JVM languages are able to target only two out of three goals. Existing AOT compilers offer compatible implementation with quick start-up time at the cost of the lower peak performance. Existing JIT compilers offer a compatible implementation with excellent peak performance at the expense of warm-up overhead.

Domain-specific compilers generate highly specialized code for a narrow subset of the language that foregoes compatibility for performance. For example, Delite [80] domain-specific

languages restrict how side effects can be used in Scala programs, and primarily focus on the compilation of purely functional programs instead.

In comparison, Scala Native meets all three design goals, achieving instant start-up time without compromising on peak performance or language compatibility.

1.2 Related Work

A significant amount of research has been done in the field of optimizing compilers for the JVM. The majority of them relies on just-in-time compilation to achieve best runtime performance.

HotSpot JVM [49] is the status quo runtime implementation used for the Scala programming language. It is implemented as a multi-tier just-in-time compiler that is focused on achieving the best peak throughput, at the expense of warm-up overhead.

The Server Compiler [64] (also known as C2) is the last tier optimizing compiler within the HotSpot JVM. It relies on the sea-of-nodes [30] intermediate representation to perform single-pass optimization of detected hot paths based on profile feedback from earlier tiers of execution.

Graal VM [86] is the next-generation optimizing compiler that aims to replace the C2 compiler. Similarly to C2, it relies on the sea-of-nodes intermediate representation [39] to perform a wide variety of optimizations [52, 53, 54, 68, 76, 85].

Graal Native Image [2] is a subproject within Graal VM that uses the same optimizing compiler to produce binaries purely ahead-of-time. Its commercial Enterprise Edition (EE) supports profile-guided optimization that is crucial to obtain the best performance results.

The Zing JVM [71] uses LLVM to replace the C2 compiler. Similarly to the Graal VM, the goal is to obtain the best peak performance after application warm-up. Moreover, Azul JVM supports state-of-the-art low-latency garbage collection [31, 82].

The GNU Compiler toolchain [77] explored ahead-of-time compilation of Java in the GCJ project [28]. The GCJ project was based on the original GCC optimizer and Boehm GC [26].

The VMKit [41] project explored building a JVM based on top of the LLVM compiler. VMKit forgoes a baseline interpreter and performs a single compilation to LLVM without adaptive reoptimization. Authors suggest that this results in suboptimal start-up performance that can be resolved through ahead-of-time compilation.

Jikes RVM [9] is a research JVM written in Java with the goal to explore adaptive optimization. As part of Jikes RVM, the MMTK [21] project is focused on exploring a wide variety of the garbage collection techniques in Java [20, 22, 23, 25, 44].

Domain-specific approaches such as LMS [72] enable specialized compilers [47, 79, 81] that

target a subset of the language features to generate highly efficient code. The subset usually does not include features such as virtual dispatch or garbage collection, which prevents its use with existing libraries that are written with the complete language in mind.

1.3 Structure

In this thesis, we study the complete design and implementation of the Scala Native optimizing compiler:

- In Chapter 3, we define NIR, an intermediate representation used throughout our compilation and optimization pipeline. NIR extends LLVM IR with support for high-level object-oriented features such as classes and traits.
- In Chapter 4, we provide an outline for a baseline compilation that relies on a simple whole-program analysis for devirtualization. We also cover the integration with garbage collection runtime and explore commix, our parallel and partially concurrent garbage collector.
- In Chapter 5, we explore a design for a flow-sensitive optimizer called Interflow. It relies on a single graph-free traversal of the whole program to perform several well-known optimizations in single optimization pass.
- In Chapter 6, we extend the Interflow with awareness of profile feedback obtained through runtime instrumentation. Moreover, we introduce white-gray code splitting as an underlying framework to reason about JIT-style speculative optimizations in the ahead-of-time setting.
- In Chapter 7, we evaluate the runtime performance of our implementation and compare it against Graal Native Image and HotSpot JVM.

1.4 Contributions

The key contributions of this thesis are:

- Design and implementation of Interflow, an optimizer that fuses a number of optimizations in a single pass over the whole program. Specifically, we focus on partial inlining and type-based approaches to propagate flow-sensitive information (Chapter 5).
- An implementation of the Scala programming language that meets three key objectives outlined above: start-up time, peak performance and compatibility. We evaluate and contrast our implementation to both existing just-in-time and ahead-of-time compilers (Chapter 7).

2 Overview

In this chapter, we present an overall structure of the Scala Native's approach to compilation. In particular, we provide a high-level overview of the different compilation modes and discuss how they connect the individual pieces into a bigger picture.

2.1 Compilation Model

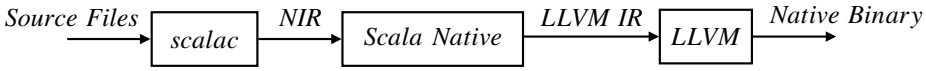


Figure 2.1 – Scala Native Compilation.

At its core, Scala Native is an ahead-of-time optimizing compiler that compiles whole programs under a closed-world assumption (Figure 2.1). Scala Native builds upon the LLVM [51] compiler infrastructure to generate platform-dependent machine code.

We distinguish three key steps:

1. **Compilation**. We rely on a modified version of the Scala compiler that emits our intermediate representation called NIR instead of JVM bytecode. Similarly to the reference implementation, compilation is incremental and supports separate compilation.

2. **Scala Native Link-Time Optimization**. Scala Native toolchain takes a set of NIR files and an application entry point as input for link-time optimization. This step is performed in batch fashion and runs the whole compilation and optimization on the subset of the NIR reachable from the entry point on each compilation.

3. **LLVM Link-Time Optimization**. After Scala Native optimizations, we emit lowered LLVM IR code and run another set of optimizations using LLVM's link-time optimizer. As the final result, we obtain an optimized statically linked binary that includes all of the application code in addition to the implementation of the garbage collector.

As we are going to elaborate in further chapters. The Scala Native distinguishes several compilation modes: baseline compilation, flow-sensitive optimized compilation, and profile-guided optimized compilation.

Each subsequent mode offers more advanced optimization pipeline, and, as a consequence, better runtime performance.

2.2 Baseline Compilation

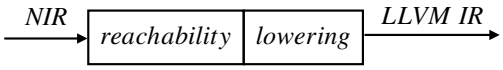


Figure 2.2 – Baseline Compilation.

The baseline compilation offers the most basic form of compilation supported by the Scala Native. Rather than targeting the best possible runtime performance, we aim to perform the basic compilation as fast as possible. This is useful for development workflows that rely on frequent recompilations.

Baseline compilation consists of two steps:

1. **Reachability**. This step aims to minimize the application size by performing whole-program reachability analysis, starting from the application entry point. Only methods and classes that are reachable are used for later stages of the compilation. In addition, this step also computes Class-Hierarchy Information (CHA) [33] that is used in later stages of compilation.

2. **Lowering**. The goal of lowering stage is to translate high-level NIR instructions into their equivalent low-level form in LLVM IR. Lowering takes advantage of CHA information to perform devirtualization based on whole-program knowledge.

Once the application is translated to the LLVM IR, is further optimized and then linked with our runtime implementation of the garbage collector (Chapter 4).

In addition to the compiled code, an application requires an implementation of a garbage collector to link against. Scala Native support four garbage collection strategies out of the box: No GC, Boehm GC, Immix GC, and Commix GC.

To obtain the best runtime performance, we link statically against the garbage collector implementation. Moreover, we take advantage of LLVM's LTO to optimize across the boundary between the application code and the application interface (Section 4.4).

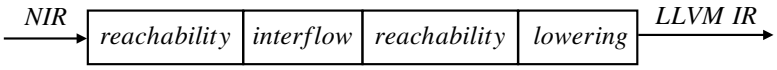


Figure 2.3 – Compilation with Interflow.

2.3 Flow-sensitive Optimization

In addition to the optimizations performed by the LLVM toolchain, we implemented our flow-sensitive optimizer called Interflow. It performs a number of optimizations in a single graph-free traversal of the whole-program starting from the application entry point (Chapter 5).

As a result of optimization, we obtain a modified program that may have made some of the methods to be unreachable due to optimizations such as inlining. Moreover, Interflow may create new methods based on the existing ones using a technique called method duplication (Section 5.5).

To account for these changes, we run another pass of reachability analysis. The optimized version of the program is then further lowered to the LLVM IR, similarly to the baseline compilation.

2.4 Profile-guided Optimization

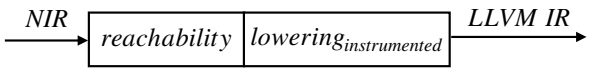


Figure 2.4 – Compilation with Profile Instrumentation.

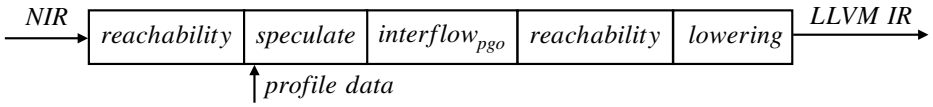


Figure 2.5 – Compilation with Profile-Guided Speculation.

To provide additional information for the optimizer, we implement an additional compilation mode that performs an alternative lowering scheme (Figure 2.4) that includes profile instrumentation.

The instrumented program is run on a workload that performs the application on a representative workload. Instrumentation records information about application runtime into a dedicated file on-disk.

The stored information is then used by the profile-guided optimization pipeline that may speculate on the invariants within a program (Figure 2.5). Moreover, the speculation step transforms the program by inserting optimistic speculative optimizations and splits off all unlikely code paths using white-gray code splitting (Chapter 6).

2.5 Conclusion

In this chapter we went through a brief overview of the Scala Native compilation pipeline. We proceed by providing a more in-depth dive into NIR, an intermediate representation used in our pipeline in the Chapter 3.

3 Native Intermediate Representation

In the previous section, we have walked through the overall design of the compilation, reachability, and optimization phases in the Scala Native pipeline. Even though we are interested in compiling Scala code, the majority of the pipeline is completely oblivious of the source language semantics and uses an intermediate language better suited for this purpose called NIR.

Some of the key features of our intermediate representation are:

- Typed object-oriented representation in SSA form.
- Equivalent in-memory and lossless textual representation.
- Stable serialized binary format used for artifact distribution.
- Hardware-independent evaluation semantics.
- Type system used to verify well-formedness of the NIR programs.

The intermediate representation is used throughout our pipeline. Binary artifacts (i.e., libraries) compiled for Scala Native are distributed as serialized NIR. Reachability, optimization and lowering phases consume and produce NIR as its only format.

NIR is designed and implemented primarily with Scala in mind although it shares lots of features common with other intermediate languages such as Java bytecode and the Swift intermediate language.

3.1 Introduction

Let us have a look at a simple Scala program that prints a countdown from 10 to 1 that is followed by a "Tada!" message:

```
1  object Countdown {
2    def main(args: Array[String]): Unit = {
3      (10 to 1 by -1).foreach { i =>
4        println(i)
5      }
6      println("Tada!")
7    }
8  }
```

The code iterates over a sequence of numbers using idomatic language features such as Scala collection ranges and closures. This code compiles down to the following NIR files on disk:

```
.
├── Countdown$$anonfun$main$1.class
├── Countdown$$anonfun$main$1.nir
├── Countdown$.class
└── Countdown$.nir
```

Each NIR file corresponds to a single JVM class file which may represent either a top-level class definition in the source program or any of the classes generated behind the scenes by the Scala compiler. For example here we see an additional class `Countdown$$anonfun$main$1` generated for the closure used to iterate over the range.

We can visualize those files by transforming them into their equivalent textual representation (method bodies omitted):

```
// Countdown$$anonfun$main$1.nir
module @"T10Countdown$"
  : @"T16java.lang.Object"
def @"M10Countdown$RE"
  : (@"T10Countdown$") => unit { ... }
def @"Countdown$D4mainLAL16java.lang.String_uE"
  : (@"T10Countdown$", array[@"T16java.lang.String"]) => unit { ... }

// Countdown$$anonfun$main$1.nir
class @"T25Countdown$$anonfun$main$1"
  : @"T39scala.runtime.AbstractFunction1$mcVI$sp", @"T18scala.Serializable"
def @"M25Countdown$$anonfun$main$1RE"
```

```
  : (@"T25Countdown$$anonfun$main$1") => unit { ... }
def @"M25Countdown$$anonfun$main$1D13apply$mcVI$spiuE"
  : (@"T25Countdown$$anonfun$main$1", int) => unit { ... }
```

NIR class files have a flat scope that contains named definitions inside of it. Definitions may not be nested within one another syntactically and may only appear at the top level of the class file.

Definitions are identified by their globally unique names (@"..."). Names encode the ownership information between methods and classes they belong to. As we are going to illustrate in Section 3.2.3, names have a rich structure that also includes types to uniquely identify overloaded definitions.

Let us have a look at the main method in detail:

```
def @"M10Countdown$D4mainLAL16java.lang.String_uE"
  : (@"T10Countdown$", array[@"T16java.lang.String"]) => unit {
%3(%1 : @"T10Countdown$", %2 : array[@"T16java.lang.String"]):
  %4 = moduleload @"T22scala.runtime.RichInt$"
  %5 = moduleload @"T13scala.Predef$"
  %6 = resolvemethod %5, #"D10intWrapperiiE"
  %7 = call[...] %6(%5, int 10)
  %8 = resolvemethod %4,
       #"D13to$extension0iiL42scala.collection.immutable.Range$InclusiveE"
  %9 = call[...] %8(%4, %7, int 1)
  %10 = resolvemethod %9, #"D2byiL32scala.collection.immutable.RangeE"
  %11 = call[...] %10(%9, int -1)
  %10 = classalloc @"T25Countdown$$anonfun$main$1"
  %11 = call[...] @"M25Countdown$$anonfun$main$1RE"(%10)
  %12 = resolvemethod %9, #"D14foreach$mVc$spL15scala.Function1uE"
  %13 = call[...] %12(%9, %10)
  %14 = moduleload @"T13scala.Predef$"
  %15 = resolvemethod %14, #"D7printlnL16java.lang.ObjectuE"
  %16 = call[...] %15(%14, "Tada!")
  ret %16
}
```

Methods are composed of basic blocks that contain instructions. Each instruction represents a single static assignment that binds a local name to a result of the right-hand side. Operations may depend on previously computed results but can never contain any nested subexpressions directly.

As we can see from the printout of the `main` method, the original code got compiled to a sequence of module accesses, class allocations and method calls on them. Methods are resolved based on their method signatures (`#"..."`). Closures are compiled to anonymous classes that have an `apply` method as an entry point for the closure invocation.

The method contains only a single basic block %3 due to the fact that looping is performed in the foreach implementation that invokes the closure for every element of the underlying collection (i.e., range of numbers in this case).

3.2 Language Definition

In this section, we are going to walk through the complete list of features that are available in the intermediate language. In particular, we are going to focus on its instruction set and how it is used to model high-level language features.

3.2.1 Programs

As we have illustrated earlier, NIR is persisted to disk in a very similar manner to the JVM class files. Each class and all their members are stored to a corresponding file in a directory structure that encodes packages names in a hierarchical manner.

JVM resolves classes through a class loading mechanism that happens at application runtime. It can be used to dynamically load arbitrary code at any point of the application execution. This allows for some extremely powerful use cases such as runtime code generation.

On the other hand, Scala Native is an ahead-of-time compiler and requires a list of all classes and methods to be statically known. While reference implementation supports runtime class loading, we perform an equivalent of class resolution at application link time. We perform this resolution as part of our reachability analysis.

Reachability analysis tries to minimize the number of classes and methods loaded from the classpath, starting from the application entry point. It traverses the complete program starting from the application entry point. As a result, we obtain a transitive closure of all reachable definitions that form a single global scope that is used for optimization and compilation to native code.

We refer to the transitive closure of reachable definitions as a *linked program*. Linked programs are self-contained units of compilation from the Scala Native point of view. They may still contain references to externally defined methods over the C ABI, but all of the Scala code is known statically.

$d ::=$	*definitions:*
$\overline{a}\ d_{top}$	*top-level definition*
$\overline{a}\ d_{member}$	*member definition*

$a ::=$	*attributes:*
`mayinline` |	*inlining attributes*
`inlinehint` |	
`noinline` |	
`alwaysinline`	
`mayspecialize` |	*specialization attributes*
`nospecialize`	
`extern`	*extern attribute*
...	

$d_{top} ::=$	*top-level definitions:*
`class` $n : n_1, \overline{n_2}$	*class definition*
`module` $n : n_1, \overline{n_2}$	*module definition*
`trait` $n : \overline{n_{n_2}}$	*trait definition*

$d_{member} ::=$	*member definitions:*
`var` $n : T = v$	*var definition*
`const` $n : T = v$	*const definition*
`decl` $n : T$	*method declaration*
`def` $n : T\ \{\ \overline{bb}\ \}$	*method definition*

Figure 3.1 – NIR Definitions.

3.2.2 Definitions

Linked programs in NIR are represented as a sequence of definitions $\overline{d}$ (Figure 3.1). Definitions provide information about available types, methods, and fields.

Type definitions d_{top} can be either classes, modules, or traits. Class and module definitions always have a single parent class they inherit from. The only exception is `java.lang.Object` that sits at the top of the class hierarchy. Additionally, they may also implement traits. Classes and traits closely match JVM bytecode classes and interfaces. Modules, on the other hand, are Scala-specific and allow one to concisely represent lazily initialized top-level state. In contrast, JVM relies on static initialization which is not supported in the Scala language.

Member definitions d_{member} correspond to either fields or methods in the original program. Methods of abstract classes and traits can be declared without implementation and are resolved at runtime through dynamic dispatch.

Definitions may be annotated with attributes a. Attributes convey additional information such as if a method can be considered for inlining and specialization by the optimizer. They also provide a way to mark special-purpose definitions such as stubs and proxies.

Lastly, attributes can also indicate that a given definition must be available externally via C ABI. Members of `extern` modules have special semantics because they map to C's top-level functions and global variables. Their names are preserved without name mangling in the native binary code.

n ::=	*global names:*
$@id$	*mangled*
`top` id	*top-level*
`member` n, s	*member*
id ::=	*identifiers:*
nv	*numeric value*
"..."	*non-empty string*

s ::=	*member signatures:*
$\#id$	*mangled*
`field` id, T	*field*
`ctor` $\overline{T}$	*constructor*
`method` $id, \overline{T}$	*method*
`proxy` $id, \overline{T}$	*reflective proxy*
`extern` id	*externally visible*
`generated` id	*internally generated*
`duplicate` $s, \overline{T}$	*method duplicate*

Figure 3.2 – NIR Names and Member Signatures.

3.2.3 Names, Signatures and Scoping

All definitions in NIR are identified by their globally unique names n (Figure 3.2). Names can be either top-level names or member names.

Top-level names are used to identify classes, modules, and traits. They contain the fully-qualified name of the corresponding definition that includes its package such as `scala.Tuple2`. Top-level names are *owners* of any of its member names. This naturally corresponds to the syntactic nesting of members definitions within class definitions in the Scala language.

Member names are composed of their owner top-level name and a member signature s. Signatures refer to fields, methods, constructors, and special-purpose members (such as internally generated and proxy names). Signatures contain types to uniquely identify a specific constructor, method overload or a method duplicate. As we are going to discuss Chapter 5, methods may be duplicated to specialize them to a sequence of more precise types than the ones declared originally.

Conceptually, member names are a part of their owner *scope* and can be looked up by their signature at either compile time or at run time through dynamic method dispatch. Owner scope contains all of its member definitions and member definitions of all of its transitive superclasses. Subclasses may override previously defined methods with the same signature.

Although names and signatures have a rich structure, they are predominantly kept in their mangled form to speed up their use as keys of hash maps and elements of hash sets. Name mangling maps them to a unique string with a similar naming convention to C++ Itanium ABI [1]. Mangling is bidirectional and can fully recover the underlying name structure without loss of information. Extern names are the only names that are not mangled in the compiled native code for the sake of interoperability with C code.

$bb ::=$	*basic block*	$t_{jump} ::=$	*jump terminators:*
$l_{bb}(\overline{l_{param} : T}):$	*label with parameters*	`jump` $l(\overline{v})$	*unconditional jump*
$\overline{l_{op} = op}$	*static assignments*	`if` v	*conditional jump*
t	*terminator*	`then` $l_1(\overline{v_1})$	
		`else` $l_2(\overline{v_2})$	
$t ::=$	*terminators:*	`switch` v	*switch jump*
$t_{break-out}$	*break-out terminators*	$\overline{\texttt{case } v_0 \Rightarrow l_1(\overline{v_1})}$	
t_{jump}	*jump terminators*	`default` $\Rightarrow l_2(\overline{v_2})$	
		`try` op	*try or else unwind*
$t_{break-out} ::=$	*break-out terminators:*	`to` $l_1 \Rightarrow l_2(\overline{v_1})$	
`ret` v	*return*	`unwind` $l_3 \Rightarrow l_4(\overline{v_2})$	
`unreachable`	*unreachable*		
`throw` v	*throw an exception*	$l ::= \%id$	*local name*

Figure 3.3 – NIR Instructions.

3.2.4 Instructions and Control-Flow

Method bodies are represented as a sequence of basic blocks (Figure 3.3). Basic blocks bb are composed of the sequence of static assignments that end with a single terminator instruction. Static assignments $l = op$ bind a local name to the result of an operation. Operations op define a data-flow within the method.

NIR uses basic block parameters to model values that depend on the control-flow rather than phi instructions [70]. Each basic block may declare a sequence of named parameters of given types. Jumps within the method body must pass values of the correspondings parameter types for a given basic block.

Terminator instructions t define control-flow transfer between basic blocks. They can transfer the control either within the method (e.g., conditional or unconditional jumps) or outside of the method (return, throw or undefined). Jumps can only be performed to the other basic blocks within the same method.

Additionally, apart from explicit control-flow through terminators, operations may cause implicit early termination of the method call due to an uncaught exception. A special terminator `try` can be used to explicitly catch and process exceptional conditions. Try terminator conditionally defines either a successful result of the normal computation or produces an exception value.

Based on the syntactic structure, it is trivial to recover a control-flow graph out of it. Even though we can represent arbitrary control-flow in the language, we are only interested in supporting *reducible* control-flow graphs [8].

Given that Scala does not have support for unstructured control-flow such as $goto$, there exists an NIR construction algorithm that preserves the reducibility property. This restriction

$op ::=$	*operations:*
op_{hl}	*high-level operations*
op_{ll}	*low-level operations*
op_{prim}	*primitive operations*

$op_{hl} ::=$	*high-level operations:*
`var`$[T]$	*declare a variable*
`varload` v	*read from variable*
`varstore` v_1, v_2	*write to a variable*
`moduleload` n	*get module instance*
`classalloc` n	*allocate an instance*
`fieldload` v, s	*readd from field*
`fieldstore` v_1, s, v_2	*write to field*
`resolvemethod` v, s	*resolve method*
`isinstanceof`$[T]$ v	*instance check*
`asinstanceof`$[T]$ v	*checked cast*
`arrayalloc`$[T]$ v	*array allocation*
`arraylength` v_1	*get array length*
`arrayload` v_1, v_2	*read from array*
`arraystore` v_1, v_2, v_3	*write to array*

$op_{ll} ::=$	*low-level operations:*
`element`$[T]$ v_0, $\overline{v}$	*get element pointer*
`stackalloc`$[T]$ v	*stack allocation*
`load`$[T]$ v	*load from memory*
`store`$[T]$ v_1, v_2	*store into memory*
`call`$[T]$ $v_0(\overline{v})$	*function call*
`insert` v_1, v_2, nv	*insert element*
`extract` v_1, nv	*extract element*

$op_{prim} ::=$	*primitive operations:*
$op_{comp}[T]$ v_1, v_2	*compare two values of type T*
$op_{conv}[T]$ v	*convert to T using conversion*
$op_{bin}[T]$ v_1, v_2	*binary operation on T*

$op_{comp} ::=$	*comparison operations:*
`ieq` | `ine`	*integer comparison*
`ugt` | `uge` | `ult` | `ule`	*unsigned integer comparson*
`sgt` | `sge` | `slt` | `sle`	*signed integer comparison*
`feq` | `fne` | `fgt` | `fge` | `flt` | `fle`	*floating-point comparison*

$op_{conv} ::=$	*conversion operations:*
`trunc` | `fptrunc`	*truncation*
`zext` | `sext` | `fpext`	*extension*
`fptoui` | `fptosi`	*floating point to integer*
`uitofp` | `sitofp`	*integer to floating point*
`ptrtoint` | `inttoptr`	*integer to pointer*
`bitcast`	*reinterpret cast*

$op_{bin} ::=$	*binary operations:*
`iadd` | `isub` | `imul`	*integer operations*
`sdiv` | `srem`	*signed integer operations*
`udiv` | `urem`	*unsigned integer operations*
`fadd` | `fsub` | `fmul` | `fdiv` | `frem`	*floating-point operations*
`shl` | `lshr` | `ashr` | `and` | `or` | `xor`	*bitwise operations*

Figure 3.4 – NIR Operations.

greatly simplifies the handling of control-flow in the optimization phase and is a common design goal between modern representations such as WebAssembly [73] and Graal IR [39].

3.2.5 Operations and Values

The computations are built up as a sequence of operations (Figure 3.4) that produce and manipulate values (Figure 3.5) at runtime. Operations op may take one or more values as an input and may also be parameterized by types. We distinguish high-level, low-level, and primitive operations.

Operations produce local names l as their results. Locals are used to express value dependencies between operations. Whenever one operation depends on another, it uses a previously

$v ::=$	*values:*
v_{prim}	*primitive value*
l	*local value*
n	*global value*
`zero`$[T]$	*zero initialized value*
`unit`	*unit value*
"..."	*string value*
$\{\ \overline{v}\ \}$	*struct aggregate value*
$[\ \overline{v}\ ]$	*array aggregate value*

$v_{prim} ::=$	*primitive values*
`null`	*null value*
`true` | `false`	*boolean value*
`char` nv	*16-bit unsigned integer value*
`byte` nv	*8-bit signed integer value*
`short` nv	*16-bit signed integer value*
`int` nv	*32-bit signed integer value*
`long` nv	*64-bit signed integer value*
`float` nv	*32-bit floating-point value*
`double` nv	*64-bit floating-point value*

Figure 3.5 – NIR Values.

computed result as one if its value arguments. Such dependencies can span across multiple basic blocks.

High-level operations op_{hl} represent object-oriented semantics necessary to express Scala code. They include managed allocations, field access, dynamic dispatch, checked casts and instance checks, boxing, and array operations. Additionally, we also include instructions to model local variables. Unlike unrestricted stack allocation, local variables return second-class values that can only be used within a single method and thus can never escape by construction. All of these operations are lowered to low-level subset during link-time.

Low-level op_{ll} and primitive operations op_{prim} closely model semantics of the corresponding operations from the LLVM IR. They include indirect calls, operations on aggregate values, raw memory access, stack allocation, arithmetic operations, comparisons, and conversions. Low-level operations may exhibit undefined behavior in error conditions such as dereferencing a null pointer. All primitive operations on non-pointer types, apart from `bitcast`, are fully checked at runtime and do not exhibit undefined behavior but rather throw exceptions similarly to JVM bytecode. As we are going to explore in Section 4.3, we rely on explicit checks to avoid undefined behavior.

Values v may either refer to previously computed results by their local name l, refer to definitions by their global name n or be one of the canonical constants for a given type. Local values are used to express dependencies between operations. Global values may be used to refer to runtime representations of types and implementations of methods.

3.2.6 Types

We rely on an object-oriented type system (Figure 3.6). Types are split into reference, primitive, aggregates, bottom, and second-class types.

Reference types T_{ref} may refer to either class, modules, traits or arrays. For convenience, we distinguish `unit` as a separate type even though it can be modeled as a named reference type

$T ::=$	*types:*	$T_{prim} ::=$	*primitive value types:*
T_{ref}	*reference type*	`bool`	*boolean value*
T_{bot}	*bottom types*	`char`	*unsigned integer value type*
T_{prim}	*primitive type*	`byte` | `short` | `int` | `long`	*signed integer value*
T_{aggr}	*aggregate types*	`float` | `double`	*floating-point value types*
$T_{second-class}$	*second-class type*	`ptr`	*pointer value type*
$T_{ref} ::=$	*reference types:*		
`unit`	*unit type*	$T_{aggr} ::=$	*aggregate value types:*
`array`$[T]$	*array reference type*	$[\ T \times n v\ ]$	*array value type*
n	*named reference type*	$\{\ \overline{T}\ \}$	*struct value type*
$T_{bot} ::=$	*bottom types:*	$T_{second-class} ::=$	*second class types:*
`null`	*null type*	`var`$[T]$	*variable type*
`nothing`	*nothing type*	$(\overline{T}) \Rightarrow T_{ret}$	*function type*

Figure 3.6 – NIR Types.

as well. Similarly to the JVM bytecode, `array`$[T]$ is the only type whose generic information is preserved. Otherwise, the type system is erased and does not preserve information about source-level generics for any other data structures.

Primitives types T_{prim} represent built-in data types such as booleans, fixed-width integer, floating-point numbers, and pointers. Primitives may not be used interchangeably with reference types and must be explicitly boxed whenever a value escapes to a generic location of a reference type. This includes `ptr` which represents C-style pointer values to unmanaged memory. By contrast, reference types are managed by the garbage-collector and don't need to be boxed.

Apart from primitive values, we also support aggregate value types T_{aggr} such as structs and array values that contain a statically known number of elements. Aggregates can be used to model multi-field value types in the source language (although Scala only supports single-field value classes due to the lack of support for value types on the JVM).

Lastly, we also include bottom T_{bot} and second-class types $T_{second-class}$. Bottom types contain `null` and `nothing` type. `null` is used as a universal bottom type for all reference types, while `nothing` models computations that do not normally return (i.e., either do not terminate or always fail with an exception). Second-class types such as variable types and functions types can not be passed as first-class values and are only used to store typing information for type checking of local variable operations and function calls.

3.2.7 Summary

As we have outlined in this section, NIR syntactically consists of three levels of nesting:

1. Top-level definitions $\overline{d}$.
2. Basic blocks and terminators $\overline{bb}$.
3. Static assignemnts and values $\overline{l = op}$.

We are going to rely on this structure as the backbone for all of the NIR transformations and analysis phases, including typing (Section 3.3), reachability (Chapter 4) and optimization (Chapter 5).

3.3 Typing

To ensure and verify the well-formedness of linked NIR programs, we define the following typing relationships:

- Program $\vdash \overline{d}$ and definition typing $\Sigma \vdash d$
- Basic block $\Delta, \Sigma \vdash bb \mid \overline{\Gamma_{out}}, T_{ret}$ and terminator typing $\Gamma, \Delta, \Sigma \vdash t \mid \overline{\Gamma_{out}}, T_{ret}$
- Operation $\Delta, \Sigma, \Gamma \vdash op : T$ and value typing $\Delta, \Sigma, \Gamma \vdash v : T$

3.3.1 Typing Environments

The typing relationships rely on the three contexts that model the three-levels of the structural nesting in the intermediate language:

- Top-level definition scope context Σ:

$$\begin{array}{ll} \Sigma ::= & \textit{top-level environment:} \\ \quad \emptyset & \textit{empty top-level} \\ \quad n \mapsto \overline{s : T}, \Sigma & \textit{class with members} \end{array}$$

  For any given top-level definition, Σ contains all of its members indexed by their signature s. Members include any signatures inherited from the parent class or trait. We rely on sigma to ensure the safety of field access and method resolution operations.

- Basic block context Δ:

$$\begin{array}{ll} \Delta ::= & \textit{block environment:} \\ \quad \emptyset & \textit{empty blocks} \\ \quad l \mapsto \overline{T}, \Delta & \textit{block with arguments} \end{array}$$

Δ stores all available basic blocks and their corresponding parameters. We rely on it to ensure that jumps within a sequence of basic block are self-contained and respect the parameter types.

- Local context Γ:

$$\begin{array}{ll} \Gamma ::= & \textit{locals environment:} \\ \emptyset & \textit{empty locals} \\ l \mapsto T, \Gamma & \textit{local of type} \end{array}$$

Γ stores all of the current accessible local names and their corresponding types. As we are going to illustrate later, Γ needs to be managed carefully to reflect the flow-sensitive naming in NIR.

3.3.2 Program and Definition Typing

NIR programs consist of definitions indexed by their unique global names. Definition typing ensures that each definition is defined only once:

$$\frac{noDoubleDefinition(\overline{d}) \quad \Sigma = \overline{n_d \mapsto members(n_d)} \quad \Sigma \vdash \overline{d}}{\vdash \overline{d}} \qquad \text{(DT-PROGRAM)}$$

Individual top-level definitions must not form cyclic dependencies in their inheritance chains:

$$\frac{n \notin parents(n) \quad isClass(n_1) \quad isTrait(\overline{n_2})}{\Sigma \vdash \texttt{class}\ n\ :\ n_1, \overline{n_2}} \qquad \text{(DT-CLASS)}$$

$$\frac{n \notin parents(n) \quad isClass(n_1) \quad isTrait(\overline{n_2})}{\Sigma \vdash \texttt{module}\ n\ :\ n_1, \overline{n_2}} \qquad \text{(DT-MODULE)}$$

$$\frac{n \notin parents(n) \quad isTrait(\overline{n_1})}{\Sigma \vdash \texttt{trait}\ n\ :\ \overline{n_1}} \qquad \text{(DT-TRAIT)}$$

Lastly, we ensure that members are well-typed with respect to their declared signatures:

$$\frac{\emptyset, \emptyset, \Sigma \vdash v\ :\ T \quad isFirstClassType(T) \quad n_{owner} = owner(n) \quad isClass(n_{owner}) \vee isModule(n_{owner})}{\Sigma \vdash \texttt{var}\ n\ :\ T = v} \qquad \text{(DT-VAR)}$$

$$\frac{\emptyset, \emptyset, \Sigma \vdash v\ :\ T \quad isFirstClassType(T) \quad n_{owner} = owner(n) \quad isModule(n_{owner})}{\Sigma \vdash \texttt{const}\ n\ :\ T = v} \qquad \text{(DT-CONST)}$$

$$\frac{T_{sig} = typesig(n) \quad \overline{T} \Rightarrow T_{ret} <: T_{sig} \quad isFirstClassType(\overline{T}) \quad isFirstClassType(T_{ret}) \quad n_{owner} = owner(n) \quad isClass(n_{owner}) \vee isTrait(n_{owner})}{\Sigma \vdash \texttt{decl}\ n : \overline{T} \Rightarrow T_{ret}} \text{(DT-DECL)}$$

$$\frac{T_{sig} = typesig(n) \quad \overline{T} \Rightarrow T_{ret} <: T_{sig} \quad isFirstClassType(\overline{T}) \quad isFirstClassTypeType(T_{ret}) \quad isReducible(\overline{bb}) \quad isReachable(\overline{bb}) \quad noDoubleDefinition(\overline{bb}) \quad \Delta = \overline{l_{bb} \mapsto paramtypes(bb)} \quad \Delta, \Sigma \vdash \overline{bb} \mid \Gamma, T_{ret}}{\Sigma \vdash \texttt{def}\ n : \overline{T} \Rightarrow T_{ret}\ \{\ \overline{bb}\ \}} \text{(DT-DEFN)}$$

3.3.3 Basic Block and Terminator Typing

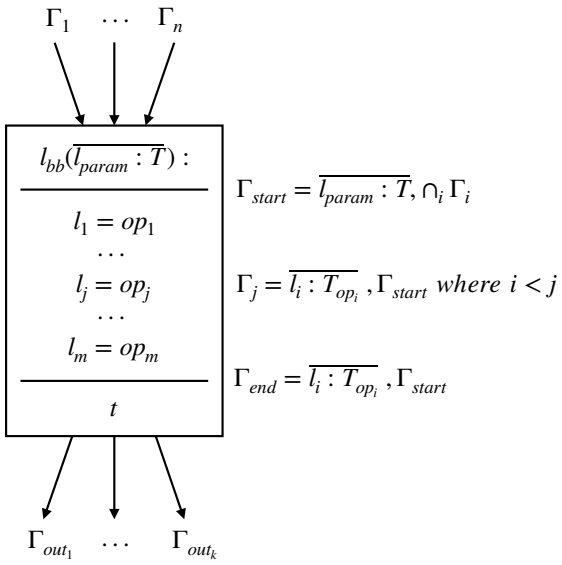


Figure 3.7 – Flow-sensitivity of the Γ environment.

The most challenging part of the whole framework lies in the basic block typing and management of the Γ context. The complexity has to do with flow-dependant nature of the local name bindings.

In SSA-based representation, instructions may depend on previously computed results as long as the dependency respects the domination property of the control-flow graph [70]. To encode domination in our typing rules, we first consider that we are working purely with reducible control-flow graphs. Such graphs may always be partitioned into forward and backward edges, where the start of the backward edge is dominated by its end.

Under this constraint, it is sufficient only to consider the DAG subset of the graph that is formed based purely on the forward edges. Before visiting a basic block, we visit all of its DAG

predecessors and then intersect the resulting Γ contexts. This ensures that the starting context Γ_{start} contains purely the bindings that are available on each path flowing into the current basic block.

Within the basic block, operations may see all of the directly preceding results, in addition to the bindings that were propagated from the predecessors and the basic block parameters $\overline{l_{param} : T}$. In the end, basic block typing completes with providing a context for each of the outgoing edges Γ_{out_i} :

$$\frac{\begin{array}{c} isFirstClassType(\overline{T}) \\ \overline{bb_{pred}} = predecessors(bb) \\ \Delta, \Sigma \vdash \overline{bb_{pred}} \mid \overline{\Gamma_{pred}}, T_{ret} \\ \Gamma_{start} = \overline{l : T}, \cap\overline{\Gamma_{pred}} \\ \Gamma_{start}, \Delta, \Sigma \vdash \overline{l_{op} = op} : \overline{T} \\ \Gamma_{end} = \overline{l_{op} : T}, \Gamma_{start} \\ \Gamma_{end}, \Delta, \Sigma \vdash t \mid \overline{\Gamma_{out}}, T_{ret} \end{array}}{\Delta, \Sigma \vdash l_{bb}(\overline{l : T}) : \overline{l_{op} = op}\ t \mid \overline{\Gamma_{out}}, T_{ret}} \quad \text{(BT-BASIC-BLOCK)}$$

$$\frac{\begin{array}{c} \Gamma, \Delta, \Sigma \vdash op : T \\ isFirstClassType(T) \vee isVarType(T) \end{array}}{\Gamma, \Delta, \Sigma \vdash l = op : T} \quad \text{(BT-LET)}$$

$$\frac{\begin{array}{c} \forall i_n \in \overline{l_{op} = op} \\ \Gamma_n = \overline{l_m : T_m}, \Gamma\ where\ m < n \\ \Gamma_n, \Delta, \Sigma \vdash i_n : T_n \end{array}}{\Gamma, \Delta, \Sigma \vdash \overline{l_{op} = op} : \overline{T_n}} \quad \text{(BT-LET-SEQ)}$$

The basic block typing terminates with basic blocks that end with break-out terminators t_{break} that do not have any outgoing edges:

$$\frac{\Gamma, \Delta, \Sigma \vdash v : T_{ret}}{\Gamma, \Delta, \Sigma \vdash \texttt{ret}\ v \mid \{\Gamma\}, T_{ret}} \quad \text{(TT-RETURN)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : @"java.lang.Throwable"}{\Gamma, \Delta, \Sigma \vdash \texttt{throw}\ v \mid \{\Gamma\}, T_{ret}} \quad \text{(TT-THROW)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{unreachable} \mid \{\Gamma\}, T_{ret} \quad \text{(TT-UNREACHABLE)}$$

Jump terminators must ensure that the destination basic block is defined in the current method and the passed arguments correspond to the types of the corresponding basic block parameters:

$$\frac{l \in \Delta \quad \overline{T} = \Delta(l) \quad \Gamma, \Delta, \Sigma \vdash \overline{v} : \overline{T}}{\Gamma, \Delta, \Sigma \vdash \texttt{jump}\ l(\overline{v}) \mid \{\Gamma\}, T_{ret}} \quad \text{(TT-JUMP)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash v : \texttt{bool} \\ \Gamma, \Delta, \Sigma \vdash \texttt{jump}\ l_1(\overline{v_1}) \mid \{\Gamma_1\}, T_{ret} \quad \Gamma, \Delta, \Sigma \vdash \texttt{jump}\ l_2(\overline{v_2}) \mid \{\Gamma_2\} T_{ret}\end{array}}{\Gamma, \Delta, \Sigma \vdash \texttt{if}\ v\ \texttt{then}\ l_1(\overline{v_1})\ \texttt{else}\ l_2(\overline{v_2}) \mid \{\Gamma_1, \Gamma_2\}, T_{ret}} \quad \text{(TT-IF)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash v : T \quad isIntegerType(T) \quad \emptyset, \emptyset, \Sigma \vdash \overline{v_0} : T \\ \Gamma, \Delta, \Sigma \vdash \overline{\texttt{jump}\ l_1(\overline{v_1})} \mid \overline{\Gamma_1}, T_{ret} \quad \Gamma, \Delta, \Sigma \vdash \texttt{jump}\ l_2(\overline{v_2}) \mid \{\Gamma_2\}, T_{ret}\end{array}}{\Gamma, \Delta, \Sigma \vdash \texttt{switch}\ v\ \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})}\ \texttt{default} \Rightarrow l_2(\overline{v_2}) \mid \{\overline{\Gamma_1}, \Gamma_2\}, T_{ret}} \quad \text{(TT-SWITCH)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash op : T \quad isFirstClassType(T) \\ \Gamma_1 = l_1 : T, \Gamma \quad \Gamma_2 = l_3 : @"java.lang.Throwable", \Gamma \\ \Gamma_1, \Delta, \Sigma \vdash \texttt{jump}\ l_2(\overline{v_1}) \mid \Gamma_1', T_{ret} \quad \Gamma_2, \Delta, \Sigma \vdash \texttt{jump}\ l_4(\overline{v_2}) \mid \Gamma_2', T_{ret}\end{array}}{\Gamma, \Delta, \Sigma \vdash \texttt{try}\ op\ \texttt{to}\ l_1 \Rightarrow l_2(\overline{v_1})\ \texttt{unwind}\ l_3 \Rightarrow l_4(\overline{v_2}) \mid \{\Gamma_1', \Gamma_2'\}, T_{ret}} \quad \text{(TT-TRY)}$$

3.3.4 Operation typing

Operation typing expresses the expected types for each of the individual operations. Special care needs to be taken to ensure that second-class types are not used as first-class values.

High-level operations op_{hl} operate purely on reference types and can not be applied to primitive or aggregate types directly:

$$\frac{n \in \Sigma \quad isModule(n)}{\Gamma, \Delta, \Sigma \vdash \texttt{moduleload}\ n : n} \quad \text{(OT-MODULELOAD)}$$

$$\frac{n \in \Sigma \quad isClass(n)}{\Gamma, \Delta, \Sigma \vdash \texttt{classalloc}\ n : n} \quad \text{(OT-CLASSALLOC)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : n \quad s : T_s \in \Sigma(n) \quad isField(s)}{\Gamma, \Delta, \Sigma \vdash \texttt{fieldload}\ v, s : T_s} \quad \text{(OT-FIELDLOAD)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : n \quad s : T_s \in \Sigma(n) \quad isField(s) \quad \Gamma, \Delta, \Sigma \vdash v_2 : T_s}{\Gamma, \Delta, \Sigma \vdash \texttt{fieldstore}\ v_1, s, v_2 : \texttt{unit}} \quad \text{(OT-FIELDSTORE)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : n \quad s : T_s \in \Sigma(n) \quad isMethod(s)}{\Gamma, \Delta, \Sigma \vdash \texttt{resolvemethod}\ v, s : \texttt{ptr}} \quad \text{(OT-RESOLVEMETHOD)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : T_1 \quad isRefType(T_1) \quad isRefType(T_2)}{\Gamma, \Delta, \Sigma \vdash \texttt{isinstanceof}[T_2]\ v : \texttt{bool}} \quad \text{(OT-ISINSTANCEOF)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : T_1 \quad isRefType(T_1) \quad isRefType(T_2)}{\Gamma, \Delta, \Sigma \vdash \texttt{asinstanceof}[T_2]\ v : T_2} \text{ (OT-ASINSTANCEOF)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : \texttt{int} \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{arrayalloc}[T]\ v : \texttt{array}[T]} \text{ (OT-ARRAYALLOC)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : \texttt{array}[T] \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{arraylength}\ v : \texttt{int}} \text{ (OT-ARRAYLENGTH)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v_1 : \texttt{array}[T] \quad \Gamma, \Delta, \Sigma \vdash v_2 : \texttt{int} \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{arrayload}\ v_1, v_2 : T} \text{ (OT-ARRAYLOAD)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v_1 : \texttt{array}[T] \quad \Gamma, \Delta, \Sigma \vdash v_2 : \texttt{int} \quad \Gamma, \Delta, \Sigma \vdash v_3 : T \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{arraystore}\ v_1, v_2, v_3 : \texttt{unit}} \text{ (OT-ARRAYSTORE)}$$

Local variable operations may only operate on the second-class values of the $\texttt{var}[T]$ type:

$$\frac{isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{var}[T] : \texttt{var}\ [T]} \text{ (OT-VAR)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : \texttt{var}[T] \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{varload}\ v : T} \text{ (OT-VARLOAD)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v_1 : \texttt{var}[T] \quad \Gamma, \Delta, \Sigma \vdash v_2 : T \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{varstore}\ v_1, v_2 : \texttt{unit}} \text{ (OT-VARSTORE)}$$

Low-level operations perform operations on the opaque pointers into the unmanaged memory:

$$\frac{\Gamma, \Delta, \Sigma \vdash v_0 : \texttt{ptr} \quad \Gamma, \Delta, \Sigma \vdash \overline{v : \texttt{long}} \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{element}[T]\ v_0, \overline{v} : \texttt{ptr}} \text{ (OT-ELEMENT)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : \texttt{long} \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{stackalloc}[T]\ v : \texttt{ptr}} \text{ (OT-STACKALLOC)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v : \texttt{ptr} \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{load}[T]\ v : T} \text{ (OT-LOAD)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash v_1 : \texttt{ptr} \quad \Gamma, \Delta, \Sigma \vdash v_2 : T \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{store}[T]\ v_1, v_2 : \texttt{unit}} \text{ (OT-STORE)}$$

$$\frac{T_f = \overline{T} \Rightarrow T_{ret} \quad \Gamma,\Delta,\Sigma \vdash \overline{v} : \overline{T} \quad \Gamma,\Delta,\Sigma \vdash v_0 : \texttt{ptr} \quad isFirstClassType(\overline{T}) \quad isFirstClassType(T_{ret})}{\Gamma,\Delta,\Sigma \vdash \texttt{call}[T]\ v_0(\overline{v}) : T_{ret}} \text{ (OT-CALL)}$$

Aggregate operations can be used to extract and insert values out of the aggregate type using compile-time checked indexes:

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : [T \times nv_1] \quad \Gamma,\Delta,\Sigma \vdash v_2 : T \quad 0 \leq nv_0 < nv_1 \quad isFirstClassType(T)}{\Gamma,\Delta,\Sigma \vdash \texttt{insert}\ v_1, v_2, nv_0 : [T \times nv_1]} \text{ (OT-INSERT-ARRAY)}$$

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : \{\overline{T}\} \quad \Gamma,\Delta,\Sigma \vdash v_2 : T_{nv} \quad 0 \leq nv < length(\overline{T}) \quad isFirstClassType(T)}{\Gamma,\Delta,\Sigma \vdash \texttt{insert}\ v_1, v_2, nv : \{\overline{T}\}} \text{ (OT-INSERT-STRUCT)}$$

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : [T \times nv_1] \quad 0 \leq nv_0 < nv_1 \quad isFirstClassType(T)}{\Gamma,\Delta,\Sigma \vdash \texttt{extract}\ v_1, nv_0 : T} \text{ (OT-EXTRACT-ARRAY)}$$

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : \{\overline{T}\} \quad 0 \leq nv < length(\overline{T}) \quad isFirstClassType(T)}{\Gamma,\Delta,\Sigma \vdash \texttt{extract}\ v_1, nv : T_{nv}} \text{ (OT-EXTRACT-STRUCT)}$$

Lastly, primitive operations op_{prim} take values of the declared type that must be supported for the corresponding operation:

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : T \quad \Gamma,\Delta,\Sigma \vdash v_2 : T \quad isValidComp(op_{comp}, T)}{\Gamma,\Delta,\Sigma \vdash op_{comp}[T]\ v_1, v_2 : \texttt{bool}} \text{ (OT-COMP)}$$

$$\frac{\Gamma,\Delta,\Sigma \vdash v : T_2 \quad isValidConv(op_{conv}, T_1, T_2)}{\Gamma,\Delta,\Sigma \vdash op_{conv}[T_1]\ v : T_1} \text{ (OT-CONV)}$$

$$\frac{\Gamma,\Delta,\Sigma \vdash v_1 : T_1 \quad \Gamma,\Delta,\Sigma \vdash v_2 : T \quad isValidBin(op_{bin}, T)}{\Gamma,\Delta,\Sigma \vdash op_{bin}[T]\ v_1, v_2 : T} \text{ (OT-BIN)}$$

3.3.5 Value typing

Type information for locals is available through the typing context Γ:

$$\frac{l : T \in \Gamma}{\Gamma,\Delta,\Sigma \vdash l : T} \text{ (VT-LOCAL)}$$

Typing of the most values is trivial due to the fact that most values represent the canonical constants of the corresponding types:

$$\Gamma, \Delta, \Sigma \vdash n : \texttt{ptr} \quad \text{(VT-GLOBAL)}$$

$$\frac{isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash \texttt{zero}[T] : T} \quad \text{(VT-ZERO)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{unit} : \texttt{unit} \quad \text{(VT-UNIT)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{null} : \texttt{null} \quad \text{(VT-NULL)}$$

$$\Gamma, \Delta, \Sigma \vdash \text{"..."} : @\text{"}java.lang.String\text{"} \quad \text{(VT-STRING)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{true} : \texttt{bool} \quad \text{(VT-TRUE)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{false} : \texttt{bool} \quad \text{(VT-FALSE)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{char}\ nv : \texttt{char} \quad \text{(VT-CHAR)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{byte}\ nv : \texttt{byte} \quad \text{(VT-BYTE)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{short}\ nv : \texttt{short} \quad \text{(VT-SHORT)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{int}\ nv : \texttt{int} \quad \text{(VT-INT)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{long}\ nv : \texttt{long} \quad \text{(VT-LONG)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{float}\ nv : \texttt{float} \quad \text{(VT-FLOAT)}$$

$$\Gamma, \Delta, \Sigma \vdash \texttt{double}\ nv : \texttt{double} \quad \text{(VT-DOUBLE)}$$

Values of the aggregate types contain other values nested within them. We first ensure the well-formedness of each of their nested values individually:

$$\frac{\Gamma, \Delta, \Sigma \vdash \overline{v : T} \quad isFirstClassType(\overline{T})}{\Gamma, \Delta, \Sigma \vdash \{\overline{v}\} : \{\overline{T}\}} \quad \text{(VT-STRUCT)}$$

$$\frac{\Gamma, \Delta, \Sigma \vdash \overline{v} : T \quad n = length(\overline{v}) \quad isFirstClassType(T)}{\Gamma, \Delta, \Sigma \vdash [\overline{v}] : [T \times n]} \quad \text{(VT-ARRAY)}$$

Additionally, we need an additional rule to encode subtyping (Section 3.3.6) of the reference and bottom types:

$$\frac{\Gamma, \Delta, \Sigma \vdash v : S \quad S <: T}{\Gamma, \Delta, \Sigma \vdash v : T} \quad \text{(VT-SUB)}$$

3.3.6 Subtyping

Subtyping is based on the standard reflexivity and transitivity rules:

$$S <: S \qquad \text{(S-REFL)}$$

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-TRANS)}$$

Reference types T_{ref} form a subtyping relation based on their declared parents with @"$java.lang.Object$" at the top of the hierarchy:

$$\frac{isRefType(T)}{T <: @"java.lang.Object"} \qquad \text{(S-TOP)}$$

$$\frac{\texttt{class}\ n : n_1, \overline{n_2} \quad n_{parent} \in parents(n)}{n <: n_{parent}} \qquad \text{(S-CLASS-PARENT)}$$

$$\frac{\texttt{module}\ n : n_1, \overline{n_2} \quad n_{parent} \in parents(n)}{n <: n_{parent}} \qquad \text{(S-MODULE-PARENT)}$$

$$\frac{\texttt{trait}\ n : \overline{n_1} \quad n_{parent} \in parents(n)}{n <: n_{parent}} \qquad \text{(S-TRAIT-PARENT)}$$

NIR type system includes two bottom types, one for the reference type hierarchy and another one for all first-class values:

$$\frac{isRefType(T) \quad isFirstClassType(T)}{\texttt{null} <: T} \qquad \text{(S-BOT-NULL)}$$

$$\frac{isFirstClassType(T)}{\texttt{nothing} <: T} \qquad \text{(S-BOT-NOTHING)}$$

Additionally, we support subtyping for function types. Even though they are not first-class values, it is convenient to express the conformance to the method signatures:

$$\frac{\overline{T_1} <: \overline{S_1} \quad S_2 <: T_2}{\overline{S_1} \Rightarrow S_2 <: \overline{T_2} \Rightarrow T_2} \qquad \text{(S-FUNC)}$$

3.4 Related Work

NIR was designed as an intermediate language between LLVM IR [51] and Scala's typed syntax trees. This has made a significant influence on the design of the lower-level features of the intermediate representation. Low-level operations, values, constants, and types are all based on corresponding features of our target language.

Unlike LLVM IR, NIR provides a complete set of primitives necessary to implement typed object-oriented languages such as Scala, Java, or Kotlin. This includes classes and traits which closely mirror the semantics of classes and interfaces in the JVM bytecode. Unlike JVM bytecode, NIR directly uses SSA form rather than stack-based evaluation semantics. This lets us skip an intermediate step of compiling to stack-based intermediate language first and then recovering SSA equivalent out of it.

Similarly to the Swift Intermediate Language (SIL) [7], NIR uses parametrized basic blocks over LLVM's phi instructions. Parametrized basic blocks are also reminiscent of continuations in intermediate representations designed for functional programming languages such as CPS [10]. Unlike SIL, NIR offers garbage-collected runtime semantics rather than having explicit reference counting operations in the intermediate language.

Compared to SSA representations of JVM bytecode such as HotSpot's Sea of Nodes [30, 38] and Graal's IR [39], we leave graph-based representation to the optimizer state. NIR's instructions are represented in an already scheduled form unlike floating nodes in graph-based intermediate representations. This dramatically simplifies the tooling around viewing and debugging NIR transformations as we can rely on textual output that is designed to be fully isomorphic to the in-memory representation without loss of information.

Lastly, NIR shares a lot in common with Scala.js IR (SJIR) [35]. Both SJIR and NIR encode Scala's high-level semantics based on an intermediate language with an erased type system. Both representations are designed for closed-world AOT-optimizing compilers for Scala. Unlike NIR, SJIR is tree-based to simplify compilation to JavaScript which does not support non-structured control-flow. NIR includes lower-level primitives for interoperability with C, while SJIR offers tight integration with JavaScript.

3.5 Conclusion

In this chapter, we defined NIR, an intermediate representation that is used throughout Scala Native's toolchain. We presented the language structure and showed how the language could be checked for well-formedness using an erased type system. Apart from the high-level features, we also include a subset of lower-level features used for interoperability with C code that also serves as a building block for the optimizing compilation and lowering pipeline.

4 Baseline Compilation

In the previous chapter, we defined NIR, an intermediate language that we are going to use throughout the compilation and optimization pipeline.

We use baseline compilation as the foundation on which we are going to build upon in further chapters. The goal of the baseline compilation is to transform programs that consist of high-level instructions into their lower-level counterparts relying on the whole program knowledge.

In addition, we also cover the memory management runtime, which relies on conservative garbage collection. We discuss the techniques used in our implementation and illustrate how the compiled code integrates with the garbage collector.

4.1 Introduction

Scala programming language [**?**] relies heavily on virtual dispatch as the foundation for a large number of its core language features:

1. **Fields**. By default, class fields in Scala are backed by getter and setter methods that can be overridden by subclasses. This means that even the basic field access in compiled as a virtual call to the corresponding getter or setter methods.

2. **Traits**. Scala encourages library design that relies on traits with abstract methods. In addition to abstract methods, traits may also contain concrete implementations that provide a default implementation for a particular function. Those implementations may be further overridden in classes that implement the trait. Scala collections rely heavily on traits to minimize reimplementation of the same operations across a wide set of concrete collection types.

3. **Closures**. Functional programming is one of the key idioms supported by the language. Under the hood, closures are compiled as anonymous classes that extend `FunctionN` trait for a given function arity `N`. Closures represent captures as fields on the underlying

class.

4. **Implicits**. Another aspect of functional programming are type classes [42]. Scala compiler automatically resolves instances through an implicit search [61, 62] mechanism that can also synthesize new type class instances based on declarative definitions. Type classes are often encoded as traits that take self reference as an explicit argument.

In the intermediate code, all of these features end up being compiled to either class or trait virtual method dispatch. Due to the fact that even the most basic operations such as field access are compiled this way, it is crucial to *devirtualize* virtual method calls whenever possible.

A naive language implementation could always compile the virtual dispatch through the well-known runtime method dispatch techniques such as virtual function tables and row displacement dispatch tables [37]. In that case, the method dispatch cost will be prohibitive for small methods such as field accesses.

Alternatively, one can perform a whole-program analysis under a closed world assumption. Specifically, Class Hierarchy Analysis (CHA) [33] can be used to find out which methods are never overridden and can be compiled as static calls. Unlike virtual calls, static calls are eligible for inlining, which is extremely valuable for small methods even for baseline compilation.

To compute CHA, we traverse a program starting from the entry point and record *summary information* about any of the methods, classes or traits that have been visited. Any methods or classes that did not get summaries created for them are discarded as not reachable. It is beneficial to minimize the set of reachable definitions as it directly affects the precision of information available in the CHA.

4.2 Reachability Analysis

4.2.1 Summaries

One of the goals of the reachability analysis is the creation of summary information (Figure 4.1) that describes definitions in the original program. Summaries describe top-level definitions (i.e., classes, traits, and objects) and contain information about their members indexed by the member signatures.

Summaries form a cyclic directed graph that corresponds to the inheritance between classes and traits in the original program. The graph can be decomposed into a union of two acyclic graphs: one that encodes relationship from subclasses to their parent classes, and another that includes all subtypes for a given type. More concretely:

1. Class information links to their direct $parents$ it inherits from. The inheritance subgraph is acyclic and always starts with @"$java.lang.Object$" at the top of the hierarchy.

```
info ::=                                          summary information:
  classinfo n {                                   top-level definition information
    parents                                       a sequence of direct parents of the given definition
    subtypes                                      a set of subtypes that implement this class or trait
    members                                       a set of members indexed by their signature
    responds                                      a mapping from method signatures to their implementation
    allocated                                     a boolean flag that states if a defn was allocated
    called                                        a set of signatures called on this top-level defn
    ...
  }
  memberinfo n {                                  member information
    T                                             member type
    ...
  }
```

Figure 4.1 – Summary Information.

2. Each class contains a set of all of it transitive $subtypes$. Subtypes subgraph is acyclic as well and serves as a precomputed form of the inheritance chain. This set lets one answer subtyping queries in constant time through a single direct lookup in a hash set.

3. Class information contains a sequence of $members$ defined directly in this class, excluding any of the inherited members. To speed up the lookup of method resolution, we also store a mapping from signatures a class $responds$ to, to their corresponding implementation. This gives us a constant time resolution of a method implementation given a method signature for all classes.

4. We also record if a class was $allocated$ at least once throughout the program, and the sequence of all method signatures that were observed to be $called$ on this class. This information makes it possible to minimize the generation of the dispatch table information and improve the precision of the virtual-method-targets estimation by excluding classes that were never allocated.

The information stored in the definition of summaries does not contain any form of a call graph information. Instead, we rely on a graph free [56] traversal for both analysis and optimization.

4.2.2 Semantic Queries

The primary goal of the summary information is to support fast response to the following queries that are used as part of lowering and optimization:

1. Lookup of a method implementation for a given exact type and signature: $n = resolve(T, s)$.

2. Lookup for all potential method implementations for a given type and signature: $\overline{n} = targets(T, s)$. The query can be answered in a linear time with respect to to the number of subtypes for a given type T. We iterate through all subtypes that are allocated and $resolve$ the signature on them.

3. Subtyping test: $isSub(T_1, T_2)$. The query can be answered in constant time based on the $subtypes$ set.

4. Least upper bound of two types: $lub(T_1, T_2)$. For class-class and class-trait pairs, the query can be answered in constant time based on the $subtypes$ set. For trait-trait pairs, computation of the least upper bound requires a linear traversal of the inheritance chain to find a least common supertrait between the two.

4.2.3 Computing Summaries and Reachability

We compute the set of all reachable definition through a single-pass traversal that goes through the program, starting from the entry points (Figure 4.2).

The algorithm takes an environment of all available definitions (i.e., the application classpath) and a sequence of entry points. Its performs a depth-first traversal that visits definitions and updates the $queue$ with any new names discovered in the process.

As one of the side effects of the traversal, we initialize and complete the summary information for each of the definitions in the program. Each time a new class is observed, it registers itself as a subtype of its parent types.

Initially, classes start with no member definitions and have them added later on as the program goes through the operations in all methods. Whenever it sees a method dispatch operation, it queries all of its potential targets using the current class hierarchy information and adds the corresponding methods targets to the $queue$ if they have not been visited yet.

Even though the information might not be complete at any point in time (due to a potential of new classes being discovered after the current method is visited) the analysis is still sound because we record all calls as part of the class information. Whenever a new subtype is loaded, it makes sure to visit all of its method implementations for the signatures that have been called before on each of its transitive parents.

Additionally, we also track if the classes have been allocated. A bare reference to a class by name (e.g., through a method signature) does not increase the set of reachable methods until the class is seen to be allocated at least once.

As a result of the traversal, we compute a set of all summaries for all reachable definitions. Any definition that does not have a summary is not reachable from the entry point and can be safely discarded.

```
1:  queue ← ∅
2:  infos ← ∅
3:
4:  procedure REACH(env = n ↦ d, entries = n)
5:      queue = queue ∪ entries
6:      while nonEmpty(queue) do
7:          n ← pop(queue)
8:          if n ∉ infos then
9:              infos(n) = reachDefn(env(n))
10:         end if
11:     end while
12:     reachable = {n ↦ d | n ↦ d ∈ env ∧ n ∈ infos}
13:     return (reachable, infos)
14: end procedure
15:
16: procedure REACHDEFN(d)
17:     for n ∈ d do
18:         queue = queue ∪ n
19:     end for
20:     if isTopLevelDefn(d) then
21:         return newTopInfo(d)
22:     else if isFieldDefn(d) then
23:         return newMemberInfo(d)
24:     else if isMethodDefn(d) then
25:         for op ∈ basicBlocks(d) do
26:             reachOp(op)
27:         end for
28:         return newMemberInfo(d)
29:     end if
30: end procedure
31:
32: procedure REACHOP(op)
33:     if isAllocation(op) then
34:         n ← allocates(op)
35:         setAllocated(infos(n))
36:     end if
37:     if isMethodDispatch(op) then
38:         for n ∈ targets(op) do
39:             queue = queue ∪ n
40:         end for
41:         n ← receiver(op)
42:         s ← signature(op)
43:         setCalled(infos(n), s)
44:     end if
45: end procedure
```

Figure 4.2 – Reachability Algorithm

4.2.4 Class Loading

Reachability algorithm takes a mapping from names to definitions as its view of all available definitions in the program. The environment is typically represented through a classpath that contains files stored in directories or jars on disk.

Each serialized NIR file on disk corresponds to a single class in the original program. All of the definitions within that file are owned by the corresponding top-level definition. This lets us perform global name resolution in two steps: the first resolution of a corresponding file on disk, and then later resolution of a member signature within that file.

Reachability traversal lazily loads classes on the first access similarly to class loading mechanism as part of the JVM. This allows us to avoid deserialization overhead for classes that are reached in a given application.

Moreover, the presented algorithm can exclude a subset of methods or fields within a class as long as they are not visited during the traversal. Once reachability analysis is complete, such definitions are discarded even if they were previously deserialized as part of the class.

4.3 Lowering High-Level Operations

In this section, we are going to walk through the lowering process that maps high-level operations to their lower-level counterparts using the whole program knowledge we have computed during the reachability analysis.

Lowering is represented through a translation function $[\![\bullet]\!]_{lower}$. It goes through the body of each method visiting one instruction at a time and replacing all op_{hl} using combination of op_{ll} and op_{prim}.

In addition to the lowered instructions, we might also emit *guards* (4.3.10). Guards represent the in-place checks for unlikely error conditions that may share a common slow path.

4.3.1 Class Allocation and Garbage Collection Interface

Our interaction with the garbage collector is straightforward and relies on the wiring of the allocation instructions to call corresponding runtime implementation of allocation for classes and arrays:

$$
\begin{aligned}
&[\![l = \texttt{classalloc}\ n]\!]_{lower} = \\
&\quad l = \texttt{call}[...]\ @"runtime\_alloc"(n)
\end{aligned}
$$

$$
\begin{aligned}
&[\![l = \texttt{arrayalloc}[T]\ v]\!]_{lower} = \\
&\quad l = \texttt{call}[...]\ @"runtime\_arrayalloc"(T,\ v)
\end{aligned}
$$

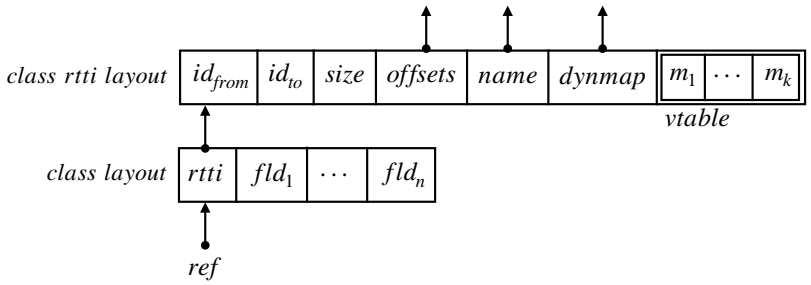


Figure 4.3 – Class and Runtime Type Information Memory Layout.

We are going to go through the details of the runtime support for the garbage collection in Section 4.4.

4.3.2 Memory Layout and Runtime Type Information

Each reachable class in the linked program is backed by a statically allocated instance of Runtime Type Information (Figure 4.3). It contains all the information necessary to reason about classes from the runtime point of view, it includes:

- **Type IDs**. All classes get unique numeric ID range that starts from id_{from} and ends with id_{to}. The unique IDs of the subclasses are nested within the ID range of its parents which allows us to represent instance tests through simple range checks. In section 4.3.5, we explain how this works with multiple inheritance.

- **Type Name**. Naming information is necessary for support a subset of the @"$java.lang.Class$" API that covers minimal runtime reflection capabilities.

- **Memory Layout Information**. This includes both the underlying class size and the offsets of all fields that store references to other objects. The memory layout information is used by the garbage collector to perform precise heap scanning.

- **Virtual Dispatch Information**. Each class is backed by a traditional virtual dispatch table and a hash map used to implement reflective calls (that correspond to calls through structural types in the original Scala program).

4.3.3 Field Access

Objects store field in a linear order. Access of the field is as simple as computing a derived pointer into the middle of the object using the statically known offset for a given field:

$$
\begin{aligned}
&[\![l = \texttt{fieldload}\ v,\ s]\!]_{lower}\ = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l_{ptr} = \texttt{element}[T_{layout}]\ v,\ \texttt{int}\ 0,\ \texttt{int}\ nv_{index} \\
&\quad l = \texttt{load}[T_{field}]\ l_{ptr}
\end{aligned}
$$

$$
\begin{aligned}
&[\![l = \texttt{fieldstore}\ v_1,\ s,\ v_2]\!]_{lower}\ = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l_{ptr} = \texttt{element}[T_{layout}]\ v,\ \texttt{int}\ 0,\ \texttt{int}\ nv_{index} \\
&\quad l = \texttt{store}[T_{field}]\ l_{ptr},\ v_2
\end{aligned}
$$

Similarly to the reference implementation, we preserve the exceptional behavior in case of null dereference. In that case, the @"$java.lang.NullPointerException$" is thrown to indicate the error condition.

4.3.4 Virtual Method Dispatch

Method resolution relies on the class summaries we have computed during the reachability analysis. The very best case for a given method call is to have at most one target. In that case, we can replace the virtual call with the statically known result:

$$
\begin{aligned}
&[\![l = \texttt{resolvemethod}\ v,\ s]\!]_{lower}\quad \texttt{if}\quad targets(T_v, s) = \{n\}\quad = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l = n
\end{aligned}
$$

Otherwise, if there are multiple potential implementations, we must fall back to runtime dispatch based on the runtime type information of a given instance. We distinguish three types of virtual method calls that all have distinct method dispatch implementations:

- **Class-Virtual**. In cases when a receiver instance is a class, we may perform virtual call through the virtual function table. This is the best case as it allows us to resolve implementation in two memory loads:

$$
\begin{aligned}
&[\![l = \texttt{resolvemethod}\ v,\ s]\!]_{lower}\quad \texttt{if}\quad isClassVirtual(T_v,\ s)\quad = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l_{rtti} = \texttt{load}[\texttt{ptr}]\ v \\
&\quad l_{ptr} = \texttt{element}[T_{rtti}]\ l_{rtti},\ \texttt{int}\ nv_{index} \\
&\quad l = \texttt{load}[\texttt{ptr}]\ l_{ptr}
\end{aligned}
$$

- **Trait-Virtual**. Method calls on instances of a trait type can not rely on virtual function tables because trait inheritance does not follow a simple hierarchical model. Instead, we rely on selector-based row displacement [37] to resolve all trait-virtual calls.

Conceptually, row displacement is a technique that aims to efficiently represent a square matrix that given a unique ID for the corresponding method signature and runtime type of the receiver.

If implemented naively, this matrix would grow quadratically as the number of methods and classes in the program. Instead, based on the observation that most cells in this matrix are in fact zero, we can compact the table by overlaying rows one on top of the other to minimize the gaps in-between.

Our implementation relies on a single global dispatch table @"$\_\_dispatch$" that is shared between all signatures called on trait instances in the program. The lowering for the method lookup needs one extra memory access indirection to resolve a method call compared to the virtual function tables:

$$
\begin{aligned}
&[\![l = \texttt{resolvemethod}\ v,\ s]\!]_{lower} \quad \texttt{if} \quad isTraitVirtual(T_v,\ s) \quad = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l_{rtti} = \texttt{load}[\texttt{ptr}]\ v \\
&\qquad \triangleright \textit{ load class pointer} \\
&\quad l_{typeid} = \texttt{load}[\texttt{int}]\ l_{rtti} \\
&\qquad \triangleright \textit{ load integer class id} \\
&\quad l_{rowptr} = \texttt{element}[\texttt{ptr}]\ @"\_\_dispatch",\ \texttt{int}\ nv_{index} \\
&\qquad \triangleright \textit{ find the the column for given method selector id} \\
&\quad l_{ptr} = \texttt{element}[\texttt{ptr}]\ l_{rowptr},\ l_{typeid} \\
&\qquad \triangleright \textit{ find the offset for given dynamic type id} \\
&\quad l = \texttt{load}[\texttt{ptr}]\ l_{ptr} \\
&\qquad \triangleright \textit{ load the method pointer from the computer offset}
\end{aligned}
$$

- **Reflective**. As the name might suggest, the reflective calls are emitted as a result of the dynamic method lookup at runtime. Their use in NIR corresponds to the method dispatch based on structural types in Scala. Unlike the other two types of method lookups, reflective calls may, in fact, fail at runtime and are not statically guaranteed to succeed.

  Given that this feature is used extremely rarely in Scala, we optimized the implementation for the smallest binary size footprint rather than the best runtime performance. The implementation uses a hash map lookup that is generated for every class and contains only the method implementations for any of the reflective calls we've observed in the linked program:

$$
\begin{aligned}
&[\![l = \texttt{resolvemethod}\ v,\ s]\!]_{lower} \quad \texttt{if} \quad isProxyVirtual(T_v,\ s) \quad = \\
&\quad \texttt{guardnotnull}\ v \\
&\quad l = \texttt{call}[...]\ @"runtime\_dyndispatch"(v,\ s) \\
&\quad \texttt{guardnosuchmethod}\ l
\end{aligned}
$$

  If the hash map lookup returns a `null` value, the reflective method lookup may fail with @"$java.lang.NoSuchMethodError$" at runtime.

4.3.5 Instance Checks and Checked Casts

Unlike all the other operations presented here, instance checks and casts are well-behaved with respect to null values. Instance checks always fail by returning a false value and casts always succeed with a null reinterpreted to the given type.

We structure the lowering of those operations in two steps. We first handle the null case on the outside and then delegate to a utility-lowering transform that assumes no nulls are possible for a given instance check or a cast:

$$
\begin{aligned}
&[\![l = \texttt{isinstanceof}[T]\ v]\!]_{lower}\ = \\
&\quad l_{notnull} = \texttt{ine}[v]\ v, \texttt{null} \\
&\quad \texttt{if}\ l_{notnull}\ \texttt{then}\ l_{check}\ \texttt{else}\ l_{result}(\texttt{false}) \\
&\quad l_{check}: \\
&\qquad [\![l_{is} = \texttt{isinstanceof}[t]\ v]\!]_{is} \\
&\qquad \texttt{jump}\ l_{result}(l_{is}) \\
&\quad l_{result}(l\ :\ \texttt{bool}):
\end{aligned}
$$

$$
\begin{aligned}
&[\![l = \texttt{asinstanceof}[T]\ v]\!]_{lower}\ = \\
&\quad l_{notnull} = \texttt{ine}[v]\ v, \texttt{null} \\
&\quad \texttt{if}\ l_{notnull}\ \texttt{then}\ l_{check}\ \texttt{else}\ l_{result} \\
&\quad l_{check}: \\
&\qquad [\![l_{is} = \texttt{isinstanceof}[t]\ v]\!]_{is} \\
&\qquad \texttt{guardclasscast}\ l_{is} \\
&\qquad \texttt{jump}\ l_{result} \\
&\quad l_{result}: \\
&\qquad l = \texttt{bitcast}[T]\ v
\end{aligned}
$$

Here, the underlying implementation $[\![l = \texttt{isinstanceof}[T]\ v]\!]_{is}$ performs the lowering of the actual runtime instance check under the assumption that the value v is guaranteed not to be not null.

In the best case, we can statically guarantee that for all subtypes of the type of the value T the instance check is going to succeed, and we can emit `true` constant as a result:

$$
\begin{aligned}
&[\![l = \texttt{isinstanceof}[T]\ v]\!]_{is}\quad \texttt{if}\quad isSub(T_v,\ T)\quad = \\
&\quad l = \texttt{true}
\end{aligned}
$$

Otherwise, if the static information is inconclusive, we need to perform the test at runtime. The implementation of the runtime test depends on the type T:

- If the type is a class type that has no subclasses we can turn the test into the check that RTTI of the value v is exactly the RTTI of the given class:

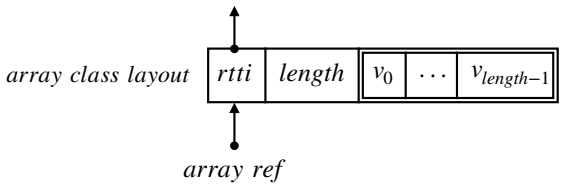


Figure 4.4 – Array Memory Layout.

$$[\![l = \texttt{isinstanceof}[n]\ v]\!]_{is} \quad \texttt{if} \quad isClass(n) \wedge |idRange(T)| = 1 \quad =$$
$$l_{rtti} = \texttt{load}[\texttt{ptr}]\ v$$
$$l = \texttt{ieq}\ [\texttt{ptr}]\ l_{rtti},\ n$$

- Otherwise, if the type is a class that has multiple subclasses, we take advantage of the nested nature of the class IDs and perform a numeric range test based on it:

$$[\![l = \texttt{isinstanceof}[n]\ v]\!]_{is} \quad \texttt{if} \quad isClass(n) \wedge |idRange(T)| > 1 \quad =$$
$$l_{rtti} = \texttt{load}[\texttt{ptr}]\ v$$
$$l_{typeid} = \texttt{load}[\texttt{int}]\ l_{rtti}$$
$$l_{ge} = \texttt{uge}[\texttt{int}]\ \texttt{int}\ nv_{id-from},\ l_{typeid}$$
$$l_{le} = \texttt{ule}[\texttt{int}]\ l_{typeid},\ \texttt{int}\ nv_{id-to}$$
$$l = \texttt{and}[\texttt{bool}]\ l_{ge},\ l_{le}$$

- Lastly, if the type T is a trait we use a statically precomputed boolean table indexed by a combination of the trait ID and the runtime type ID to lookup the boolean value stored in it:

$$[\![l = \texttt{isinstanceof}[n]\ v]\!]_{is} \quad \texttt{if} \quad isTrait(n) \quad =$$
$$l_{rtti} = \texttt{load}[\texttt{ptr}]\ v$$
$$l_{typeid} = \texttt{load}[\texttt{int}]\ l_{rtti}$$
$$l_{ptr} = \texttt{element}[T_{table}]\ @"\_\_class\_has\_trait",\ \texttt{int}\ 0,\ l_{typeid},\ \texttt{int}\ nv_{trait-id}$$
$$l = \texttt{load}[\texttt{bool}]\ l_{ptr}$$

4.3.6 Array Operations

Java-style arrays store their elements linearly in memory together with their length (Figure 4.4). The length is fixed and may not change after the array allocation.

Access of the array length closely mirrors the regular field access we have seen earlier:

$$[\![l = \texttt{arraylength}\ v]\!]_{lower} =$$
$$\texttt{guardnotnull}\ v$$
$$l_{ptr} = \texttt{element}[T_{layout}]\ v,\ \texttt{int}\ 0,\ \texttt{int}\ nv_{index}$$
$$l = \texttt{load}[\texttt{int}\ ]\ l_{ptr}$$

Access of the array elements, on the other hand, needs to additionally ensure that the array index is in fact in bounds with respect to the array length:

$$
\begin{aligned}
&[\![l = \texttt{arrayload}\ v_1, v_2]\!]_{lower} \ = \\
&\quad [\![l_{len} = \texttt{arraylength}\ v_1]\!]_{lower} \\
&\quad \texttt{guardinbounds}\ v_2,\ l_{len} \\
&\quad l_{ptr} = \texttt{element}[T_{layout}]\ v1,\ \texttt{int}\ 0,\ \texttt{int}\ nv_{index},\ v_2 \\
&\quad l = \texttt{load}[T_{elem}]\ l_{ptr}
\end{aligned}
$$

$$
\begin{aligned}
&[\![l = \texttt{arraystore}\ v_1, v_2, v_3]\!]_{lower} \ = \\
&\quad [\![l_{len} = \texttt{arraylength}\ v_1]\!]_{lower} \\
&\quad \texttt{guardinbounds}\ v_2,\ l_{len} \\
&\quad l_{ptr} = \texttt{element}[T_{layout}]\ v_1,\ \texttt{int}\ 0,\ \texttt{int}\ nv_{index},\ v_2 \\
&\quad l = \texttt{store}[T_{elem}]\ l_{ptr},\ v_3
\end{aligned}
$$

While this may incur seemingly redundant access to the array length on every element access, we can take advantage of the fact that array length never changes. A simple pass of redundancy elimination will remove all redundant memory accesses [57].

4.3.7 Primitive Operations

NIR does not rely on undefined behavior for arithmetic error conditions such as division by zero. Instead, we throw exceptions similar to the reference implementation. To support this semantics, we need to inject additional handling code in the following cases:

- The unsigned division needs to guard against the cast of a zero divisor:

$$
\begin{aligned}
&[\![l = op_{bin}[T]\ v_1,\ v_2]\!]_{lower} \quad \texttt{if} \quad op_{bin} \in \{ \texttt{urem}, \texttt{udiv} \} \quad = \\
&\quad \texttt{guardnotzero}\ v_2 \\
&\quad l = op_{bin}[T]\ v_1,\ v_2
\end{aligned}
$$

- Signed division not only needs to check for division by 0 but also needs to take into an account the possibility of signed overflow that can occur when the smallest value of given integer type is divided by -1:

$$
\begin{aligned}
&[\![l = op_{bin}[T]\ v_1,\ v_2]\!]_{lower} \quad \texttt{if} \quad op_{bin} \in \{ \texttt{srem}, \texttt{sdiv} \} \quad = \\
&\quad \texttt{guardnotzero}\ v_2 \\
&\quad \texttt{guardnodivoverflow}\ v_1,\ v_2 \\
&\quad l = op_{bin}[T]\ v_1,\ v_2
\end{aligned}
$$

- The semantics of shifts is only defined if the number of bits is less than bits in the given integer type. To account for that, we mask right-hand side by a mask $bits - 1$.

$$
\begin{aligned}
&[\![l = op_{bin}[T]\ v_1,\ v_2]\!]_{lower} \quad \texttt{if} \quad op_{bin} \in \{\, \texttt{shl}, \texttt{lshr}, \texttt{ashr} \,\} \quad = \\
&\quad l_{masked} = \texttt{and}[T]\ v_2,\ v_{bits-1} \\
&\quad l = op_{bin}[T]\ v_1,\ l_{masked}
\end{aligned}
$$

- Lastly, the floating-point to integer conversions need to handle the case of potential overflow when the resulting floating value does not fit in the range between minimum and maximum values of the resulting type T:

$$
\begin{aligned}
&[\![l = \texttt{fptosi}\ [T]\ v]\!]_{lower} \ = \\
&\quad \texttt{guardfpinrange}\ v,\ v_{T_{min}}, v_{T_{max}} \\
&\quad l = \texttt{fptosi}\ [T]\ v
\end{aligned}
$$

$$
\begin{aligned}
&[\![l = \texttt{fptoui}\ [T]\ v]\!]_{lower} \ = \\
&\quad \texttt{guardfpinrange}\ v,\ v_{T_{min}}, v_{T_{max}} \\
&\quad l = \texttt{fptosi}\ [T]\ v
\end{aligned}
$$

4.3.8 Module Initialization

NIR modules correspond directly to Scala's top-level `object` syntax. They provide an easy way to define a type that has at most one instance at runtime. We store all of those instances in a contiguous memory that is indexed by the unique ID of the module.

On first access modules needs to be allocated an initialized. Given that it happens only once during the whole application runtime, we assume that modules are always initialized on the fast path of the module access operation:

$$
\begin{aligned}
&[\![l = \texttt{moduleload}\ n]\!]_{lower} \ = \\
&\quad \texttt{guardinitialized}\ n \\
&\quad l_{ptr} = \texttt{element}[\texttt{ptr}]\ @"\_\_modules", \texttt{int}\ nv_{index} \\
&\quad l = \texttt{load}[\texttt{ptr}]\ l_{ptr}
\end{aligned}
$$

Whenever the condition is false, and a module instance at a given offset is, in fact, null, we go on a slow path that performs the initialization.

4.3.9 Local Variables

Local variables provide a constrained version of general-purpose stack allocation. In the baseline compilation, they map directly to stack allocation, and direct memory accesses to the stack-allocated memory:

$$[\![l = \texttt{var}[T]]\!]_{lower} =$$
$$\quad \texttt{stackalloc}[T]\ \texttt{int}\ 1$$

$$[\![l = \texttt{varload}\ v]\!]_{lower} =$$
$$\quad \texttt{load}[T_{var}]\ v$$

$$[\![l = \texttt{varstore}\ v_1,\ v_2]\!]_{lower} =$$
$$\quad \texttt{store}[T_{var}]\ v_1,\ v_2$$

As we are going to see later, local variables are always optimized away due to the fact that they may never escape.

4.3.10 Guards

Guard is a flexible abstraction for error handling that gets compiled as an inline check that fallbacks to a common slow path at the end of the method.

The error conditions that are verified by guards are assumed never to fail, but we still need to keep the slow path handling of the code. Let us have a look at the result of the lowering of two subsequent field accesses:

$$\begin{aligned}
&\texttt{guardnotnull}\ v_1 \\
&l_{ptr1} = \texttt{element}[T_{layout}]\ v_1,\ \texttt{int}\ 0,\ \texttt{int}\ 1 \\
&l = \texttt{load}[T_{field}]\ l_{ptr1} \\
&\texttt{guardnotnull}\ v_2 \\
&l_{ptr2} = \texttt{element}[T_{layout}]\ v_2,\ \texttt{int}\ 0,\ \texttt{int}\ 2 \\
&l = \texttt{load}[T_{field}]\ l_{ptr2}
\end{aligned}$$

The naive representation of the guard would generate a full-blown branch that has handles two possible cases for each of the accesses:

```
    l_notnull1 = ine [ptr] v_1, null
    if l_notnull1 then l_continue1 else l_fail1
  l_fail1:
    call[...] @"runtime_throw_null_deref"()
    unreachable
  l_continue1:
    l_ptr1 = element[T_layout] v, int 0, int 1
    l = load[T_field] l_ptr1
    l_notnull2 = ine [ptr] v_2, null
    if l_notnull2 then l_continue2 else l_fail2
  l_fail2:
    call[...] @"runtime_throw_null_deref"()
    unreachable
  l_continue2:
    l_ptr2 = element[T_layout] v, int 0, int 2
    l = store[T_field] l_ptr2, v_2
```

Instead, guards reuse the same slow path across all guard uses that is generated at the end of the method after all of the preexisting code:

```
    l_notnull1 = ine [ptr] v_1, null
    if l_notnull1 then l_continue1 else l_common-fail
  l_continue1:
    l_ptr1 = element[T_layout] v, int 0, int 1
    l = load[T_field] l_ptr1
    l_notnull2 = ine [ptr] v_2, null
    if l_notnull2 then l_continue2 else l_common-fail
  l_continue2:
    l_ptr2 = element[T_layout] v, int 0, int 2
    l = store[T_field] l_ptr2, v_2

...

  l_common-fail:
    call[...] @"runtime_throw_null_deref"()
    unreachable
```

This code gets compiled into compact machine code with side-exiting branches that are never taken. Moreover, the fact that the slow path code is at the end of the method minimizes the risk of it being mistakingly loaded in cache memory instead of the actual application code. So effectively, we perform an ad-hoc code positioning [67] based on the expectation that error cases are unlikely to happen.

4.4 Runtime Support for Garbage Collection

4.4.1 Design Constraints

Scala Native is built upon the LLVM compilation toolchain and inherits some of its underlying limitations. In particular, LLVM has no built-in garbage collector but only provides multiple types of extension points to integrate the compiler with an existing collector.

As one possibility, we could have used those intrinsics to build a precise garbage collector. Based on the initial investigation we have observed that the garbage collector interface is immature and is not equally supported on all platforms. For example, as of this writing, only x86_64 architecture is fully supported by the statepoint intrinsics [4]. While there is nothing fundamental in those restrictions, the perceived cost of adopting those intrinsics was considered to be high.

Instead of building upon those intrinsics, we have decided to limit ourselves to the *conservative* garbage collectors [18, 26, 75]. A conservative collector is able to reclaim memory without having the exact information about all of the memory locations. Such collectors treat values as ambiguous references and conservatively classify them as a potential heap reference if they are within the valid memory range.

As a positive benefit of the conservative root scanning, we can freely share Scala objects between the runtime implemented in both C and Scala without having to have an additional indirection such as object handles.

As a downside to this decision, the GC is not allowed to move objects referenced from the stack. While this may limit our design space, this did not turn out to be a significant obstacle to getting good runtime performance on our case study projects (Chapter 7).

Scala Native supports four distinct memory management strategies that all satisfy the conservative root scanning restriction: No GC, Boehm GC, Immix GC, and Commix GC.

4.4.2 No GC

The simplest form of memory management that satisfies all of our design constraints is a toy garbage collector that never reclaims any memory.

The allocator of this GC bump allocates across a large (4G) preallocated memory area. Whenever the area is exhausted, it tries to claim another one from the operating system. The allocator fails as soon as it can not claim any more memory.

While this collector is not practical for most applications, it offers the benefit of being a good baseline to benchmark against. It effectively incurs close no runtime overhead as it never has to reclaim any memory. Moreover, it uses the best-known allocation strategy that offers the best mutator performance.

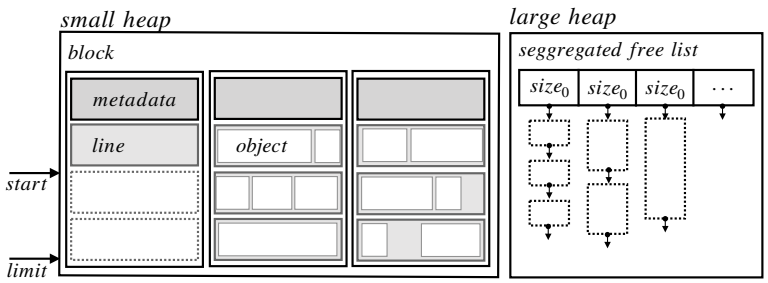


Figure 4.5 – Scala Native's Immix Heap Layout

4.4.3 Boehm GC

The collector by Boehm and Weiser [26] is the most widely used conservative garbage collector. It offers a simple garbage collector interface that can also be used as a leak detector for manual memory managed code.

At its core, Boehm GC is a mark-and-sweep garbage collector. It provides support for incremental and generational garbage collection. Our integration with Boehm relies on the fully conservative variant of the API.

4.4.4 Immix GC

Lukas Kellenberger [6] implemented a collector that closely follows the design of the original work by Blackburn and McKinley [24]. As suggested by Shahriyar et al. [75], Immix is an excellent foundation for conservative garbage collectors.

Our implementation features:

- **Small and Large Heap**. Heap is structured into two separate areas for small and large allocations. Small heap uses Immix algorithm, while the large heap uses mark-and-sweep with segregated free lists to support allocations that do not fit in a single block.
- **Block-Based Heap Structure**. The small heap is structured as a sequence of blocks (32K) that are composed of lines (256B). Objects may span multiple lines but can not span blocks. Each block starts with an object header that stores metadata information about the block itself in addition to metadata information for each of the lines within it.
- **Bump Allocation**. Allocator performs bump allocation to allocate within a contiguous sequence of free lines. Medium-size objects are backed by an additional overflow allocator that allocates purely in completely free blocks. The split between the two allocators allows one to reduce the risk of space being wasted due to interleaving of medium and small allocations.

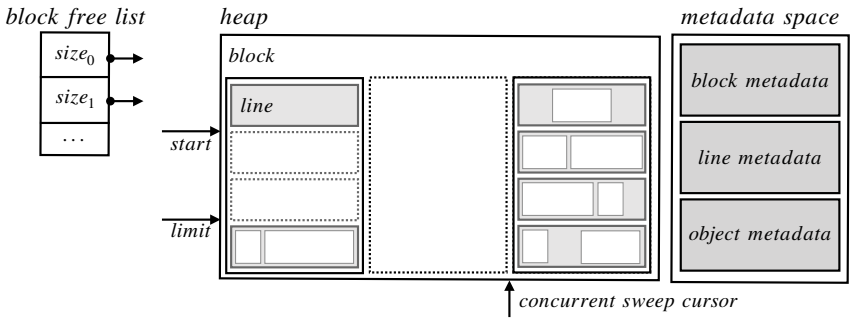


Figure 4.6 – Commix Heap Layout

- **Single-Threaded Stop-The-World Collection**. The implementation reclaims memory through a single-threaded stop-the-world collection that happens on the mutator thread. It consists of marking and sweeping phases and may reclaim free lines and blocks. Given that allocation happens on the per-line boundary, we do not reclaim memory on a finer-grain level than a single line.
- **Non-Moving Collection**. Unlike the original Immix design [24] and the conservative variation of it [75], our implementation is completely non-moving. Object's location in memory is fixed at allocation time and may not change until it has been reclaimed.

4.4.5 Commix GC

Valdis Adamsons [3] refined our original Immix implementation with support for parallel and partially concurrent garbage collection. Besides, the implementation got rid off the split between large and small heaps and changed the layout of the GC metadata.

The implementation builds upon the existing implementation with the following improvements:

- **Unified Heap**. Rather than relying on separate small and large heaps, we unify it to contain both small and large objects. Blocks are managed using segregated free lists and can be claimed both for small object allocation and large allocation that can span multiple blocks.
- **Seggregated Metadata**. The metadata is stored in a separate space that contains additional information about blocks, lines, and objects. This ensures that the heap is dedicated purely to the application memory. The change was originally motivated as part of the implementation to support the unified heap, but we later found out that it also improved runtime performance on our case study workloads.
- **Parellel Stop-The-World Marking**. Commix performs stop-the-world parallel marking that scales up to 8 hardware threads. It uses gray packets [16, 63] as the means to

schedule the marking across multiple threads.

Each thread takes a fixed-size packet that contains a sequence of objects to be visited and may produce one or more packets as a result. The marking phase terminates when there are no more packets left to process.

Due to the fact that work per object is not equal, we may split packets to avoid skewed workloads. Arrays of objects are the typical example of a single work item that may contain an amount of work that is not comparable to a single small object.

Work scheduler automatically spins the threads up based on the current number of available gray packets. Due to the breadth-first nature of the marking implementation, we may start with a few packets that grow as more of the heap is visited. The number of threads will grow to accommodate the gradually increasing amount of work.

- **Parallel Concurrent Sweeping**. The sweeping phase visits the heap in linear order. Naturally, the amount of work scales linearly with the increase in the heap size. Commix removes the heap size as the factor for the garbage collection pause time by performing sweeping concurrently with the application.

  The heap is swept by multiple worker threads that traverse batches of blocks and reclaim the processed blocks back to the block free list. Special care needs to be taken for free blocks on the batch boundary due to the fact that we need to reclaim all contiguous areas together as a single memory area. Before reclaiming such blocks, we need to coalesce all contiguous memory areas. This work is always done on the first GC thread.

  Whenever marking is finished the control on the mutator thread will return back to the application code. To prevent the case when the mutator outpaces the sweeping progress, we reserve a small number of free blocks. In case the reserve is not enough, the mutator thread will take a single sweeping batch to cooperate on advancing the sweep progress and retry allocation afterward.

  Due to the fact that both sweeping threads and mutator can concurrently update the free block list, we must make sure that all of the updates a correctly synchronized.

- **Improved Allocation Hot Path**. As one of the refinements of the original implementation, we improved the allocation hot path to be fully inlined into the application code (See 4.4.6). In addition, we also take advantage of prefetching hints to make sure that memory that follows after the current bump allocation cursor is going to keep being in the cache.

As a result of these changes, Commix improves upon both mutator performance and garbage collection pause times.

4.4.6 Optimizing Across Runtime Boundary

The performance of the allocation hot path is crucial to provide mutator performance that is competitive with state of the art garbage collectors. Modern JIT compilers specialize the bump allocation hot path by injecting hand-tuned GC allocation fast path at each allocation site.

Rather than duplicating the GC implementation logic in our baseline compiler, instead we opt-in for the use of LLVM's Link Time Optimization (LTO) to optimize across the boundary between the GC implementation and the application.

This allows compiling allocations as calls into runtime the implementation without any loss of performance compared to a hand-tuned version. We annotate the hot path of the GC code using `alwaysinline` attribute to ensure it is always inlined for all callers.

4.5 Related Work

Optimization of the virtual function dispatch in the context of object-oriented languages has been studied in depth [13, 33, 37, 38]. We rely on Class Hierarchy Analysis to perform devirtualization decisions. Methods that could not be devirtualized are compiled as either virtual function tables and row displacement tables.

Scala.js [35] is an ahead-of-time compiler for Scala that emits statically JavaScripts programs optimized under the closed-world assumption. Our reachability analysis algorithm is closely related to their work. Unlike the Scala.js toolchain, we only support the non-incremental batch compilation of the whole program.

Immix [24] is the original foundation for both of our garbage collector implementations. Similarly to [75], we extend the original algorithm to add support for ambiguous root tracking. The main difference to the prior work is that our garbage collector implementations are non-moving.

4.6 Conclusion

In this chapter, we have walked through the baseline compilation model that relies on a whole-program analysis for devirtualization.

In the next chapter, we are going to illustrate how we can improve upon through an addition of a flow-sensitive optimization pass that happens in-between whole-program analysis and lowering phases.

5 Interflow: Flow-sensitive Optimization

In the previous chapter, we have walked through the baseline compilation model that takes advantage of the class hierarchy analysis to transform programs with high-level instructions into their lower-level counterparts.

While the baseline compilation model already takes advantage of the whole-program knowledge, this knowledge is coarse-grained and spans the invariants across the whole program. For example, a single method override can make all calls with the same method signature to be dispatched dynamically through tables at runtime rather than statically.

Instead of relying purely on the whole-program invariants, in this chapter, we explore a design for an aggressive flow-sensitive optimizer.

5.1 Introduction

As a motivating example, let's have a look at Scala's collection library. It relies heavily on virtual dispatch to support a single implementation of combinator methods across a wide variety of concrete collections. A typical example is the `map` method defined in the `TraversableLike` trait (Figure 5.1).

Apart from the natural polymorphism on the element type, this method is simultaneously polymorphic in:

1. The collection type of `this`. The same method implementation is used for most descendants of this trait (e.g., wrapped arrays, vectors, maps, sets). Based on the collection type, `foreach` dispatches with a new closure that wraps `f` and appends its result to the collection builder `b`.

2. In the operation `f` that is being applied to every element of the collection. Scala's closures are compiled as anonymous classes that extend `FunctionN` trait (where N is from 0 to 22 depending on the number of parameters) trait that has an `apply` method that's used

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That = {
  def builder = {
    val b = bf.apply(repr)  // virtual call to apply
    b.sizeHint(this)        // virtual call to sizeHint
    b
  }
  val b = builder
  this.foreach {            // virtual call to foreach
    x =>                    // virtual call to closure's apply
      b.+=(f.apply(x))      // 2 virtual calls per iteration: += and apply
  }
  b.result                  // virtual call to result
}
```

Figure 5.1 – Definition of the map method in Scala collections.

to invoke a closure.

3. In the builder for the resulting collection through the `bf` argument. This parameter is implicit and is typically inferred by the compiler [62]. Nevertheless, users may still provide a custom instance of `CanBuildFrom` to produce a different collection type as the result of the `map` combinator.

The whole program analysis we presented previously would not be able to optimize any of those virtual calls away by itself due to the fact that even the smallest program relies on more than one collection type and a large number of closures. As a consequence, all of the operations listed above will have to pay the price of performing dynamic dispatch at runtime. Moreover, we will not be able to optimize across the method boundary due to the fact that dynamic dispatch is an optimization barrier from an inlining point of view.

The reference implementation of Scala programming language is compiled to the JVM bytecode that runs in a virtual machine that takes advantage of just-in-time compilation. Prokopec et al. [69] studied techniques that a state-of-the-art JIT compiler such as GraalVM is going to use to optimize this method:

1. Use Class Hierarchy Analysis [33] to check if methods such as `foreach` and closure's `apply` are never overridden. For our running example, this check would inevitably fail as `foreach` is overridden for every collection (dozens of methods) and `apply` is overridden for every closure (hundreds of methods).

2. Collect type profile on all of the three sources of polymorphism: `this`, `f` and `bf` local variables in this method.

3. If a type profile shows a few dominant types, the compiler will *speculatively* optimize this method assuming it is only used for dominant types. In that case, all of the virtual

calls are going to be turned into static calls behind a guard that verifies that optimization assumptions are correct (i.e., the method is used only for the types that it was compiled for).

4. Otherwise, calls will be considered megamorphic, and all of the calls are going to be compiled as virtual calls based on table lookups. In this case, the optimizer cannot inline them, and the compiled code is going to incur a performance penalty.

In case of a megamorphic type profile, these techniques cannot optimize a given method *in isolation*. Megamorphic virtual calls are optimization barriers that prevent optimizations across the call boundary.

Inlining the whole `map` method and all of its transitive dependencies is the last resort to optimize such combinators. However, inlining is based on heuristics, so it is not guaranteed to succeed (for example if the caller is already big and does not have enough size budget to fit the result of the inlining).

In summary: the problem of optimizing away multiple nested layers of virtual dispatch has not been fully addressed. While there exist techniques that can be used towards that end, they rely on heuristics that might fail unpredictably leaving the program with the cost of unoptimized dynamic dispatch.

5.2 Intuition

We designed Interflow with the primary goal of static devirtualization in mind. Specifically, we are interested in having collection combinators (such as `map`) in the Scala standard library not to pay the cost of the megamorphic virtual dispatch.

At its core, Interflow is a single pass optimizer that fuses a number of techniques in a single graph-free free traversal over the whole program [56]. The traversal aims to partially evaluate all of the parts of the program that are known statically and additionally infer and propagate precise type information across the code that could not be partially evaluated away.

The fundamental idea behind Interflow's approach to devirtualization is method duplication guided by whole-program flow-sensitive type propagation. Rather than attempting to create a single optimized version of a collection combinator such a `map`, Interflow duplicates it per context. For example, consider multiple calls to `map` with multiple different closures:

```
1  val arr = Array(1, 2, 3)
2  arr.map(_ + 1)
3
4  val vec = Vector(1, 2, 3)
5  vec.map(_ * 2)
```

With Interflow, these two calls will be transformed to use duplicates of `map` that are specialized for a combination of the exact collection type, the exact anonymous class used to map elements and exact builder for the resulting collection:

```
1  val arr = Array(1, 2, 3)
2  arr.‘map<WrappedArray.ofInt, AnonFun1, ArrayCBF>‘(_ + 1)
3
4  val vec = Vector(1, 2, 3)
5  vec.‘map<Vector, AnonFun2, VectorCBF>‘(_ * 2)
```

The duplicate versions of `map` will have more precise parameter types than the ones provided in the original version. The process will start over within each duplicate, and the virtual calls to `foreach`, `+=` and `apply` are going to be statically routed to the corresponding implementations guided by precise types of the arguments.

The resulting code will be completely free of virtual calls on the hot path. In turn, a combination of inlining, constant propagation, and allocation sinking is going to optimize it further by removing intermediate closure and box allocations and avoiding allocation of the builder altogether.

It is important to highlight that the success of devirtualization here *does not depend on the inlining of the calls to map*. If `map` is not inlined, Interflow will have to allocate the outer closure, but all of the dispatch within the `map` is still going to be static thanks to the propagation of the caller context information to the duplicate of the implementation.

5.3 Operations

As we have illustrated in the NIR language definition, we can think of the language as having three primary levels: static assignments, basic blocks with control-flow between them, and lastly method definitions. We are going to define our optimization on each of those levels separately and build up towards whole-program optimization one level at a time.

While all of the individual optimizations are well-known and have been previously studied in isolation, our focus is to produce a single fused optimization pass that combines all of them together. As we are going to explore in Section 5.5 this enables context-sensitive inlining that considers the state of other optimizations (such as allocation sinking and code motion) as one of the key inlining incentives.

We start with a simple subset of NIR that only contains a sequence of static assignments $\overline{l = op}$. Sequences of static assignments are the core of every basic block, and they may not contain any control-flow apart from an early termination due to exceptional conditions. Later, in Section 5.4, we are going to extend this framework to support optimization across multiple basic blocks within a single method.

5.3.1 Intuition

The initial inspiration for the operation optimization rules comes from the evaluation semantics for the operations in NIR that can be formulated as a small step evaluation semantics:

$$l = op \mid \sigma \longrightarrow_{eval} l = v \mid \sigma'$$

Evaluation goes through the static assignments in order and tries to evaluate them one at a time. The final result of each side effect (be it an assignment or heap modification) is reified in the state σ.

Given that we are looking at evaluation from an optimizer point of view, we are only interested in the evaluation of exception-free operation reductions. If we can not prove the absence of the exceptional condition, the instruction must remain in the generated code to preserve side-effects. From this point of view we accept that evaluation rules might get stuck without making progress; this means that the operation is emitted at runtime and could not be optimized away.

If no evaluation rule applies (e.g., we could not prove the absence of error conditions) then the result of an evaluation of an operation is an opaque named reference $l : T$, where T is the result type of the operation. Opaque named references can also appear as basic block parameters, and we assume nothing is known about them apart from their declared type.

From the evaluation semantics point of view, the state σ can be seen as a pair (ϕ, μ):

1. $\phi = \overline{l \mapsto v}$ – models the result values for all locals we've observed so far.
2. $\mu = \overline{k \mapsto (T, \overline{v})}$ – models the state of the heap as a mapping from unique key k to the allocation state that persists runtime type T and state of all fields $\overline{v}$.

Our optimization rules are going to be formulated in a manner similar to evaluation rules that take an operation $l = op$, and a state σ as inputs and either produce a transformed operation or have it evaluated statically to a value v. Additionally, we also produce an updated state σ' that reflect any of the changes made to the state.

We are going to reuse ϕ and μ to model what is known about locals and the heap statically. Some of the optimizations might require additional information in σ (e.g. ρ introduced in Section 5.3.8) or refine the structure of the existing components such as μ (e.g. Sections 5.3.3, 5.3.6).

5.3.2 Constant Propagation

Our optimization rules are going to rely on the evaluation semantics directly. The most straightforward application of evaluation semantics is constant propagation.

Given a primitive operation that takes purely constant arguments, we can evaluate it and replace the operation with a constant value that corresponds to its result. Moreover, we would like to propagate the computed results throughout the rest of the program, which in turn can enable further operations to be evaluated statically.

As we have noted before, not all operations are safe to evaluate statically due to exceptional conditions. Therefore we are interested in *sparsely* propagating constants throughout the parts which can be optimized away and leave the rest unchanged.

We represent constant propagation as an evaluation relation $\longrightarrow_{const}$:

$$\frac{\overline{v} \in op \quad isConst(\overline{v}) \quad isPrim(op) \quad isPure(op) \quad op \mid \sigma \longrightarrow_{eval} v \mid \sigma'}{l = op \mid \sigma \longrightarrow_{const} l = v \mid \sigma'} \text{ (CP-IS-CONST)}$$

$$\frac{\overline{v} \in op \quad \neg isConst(\overline{v}) \vee \neg isPrim(op) \vee \neg isPure(op) \quad \overline{v} \mid \sigma \longrightarrow_{v} \overline{v}'}{l = op \mid \sigma \longrightarrow_{const} l = [\overline{v} \mapsto \overline{v}']op \mid \sigma} \text{ (CP-NON-CONST)}$$

Constant propagation directly invokes the evaluation semantics on all pure primitive operations in our program. The results of evaluated operations is stored in ϕ as part of the σ state.

If the operation can not be evaluated, we evaluate all of its nested values and replace them with their corresponding results from ϕ:

$$\frac{l = v' \in \phi}{l \mid \sigma \longrightarrow_{v} v'} \text{ (CV-LOCAL)} \qquad \frac{\overline{v} \mid \sigma \longrightarrow_{v} \overline{v}'}{[\overline{v}] \mid \sigma \longrightarrow_{v} [\overline{v}']} \text{ (CV-ARRAY)}$$

$$\frac{\overline{v} \mid \sigma \longrightarrow_{v} \overline{v}'}{\{\overline{v}\} \mid \sigma \longrightarrow_{v} \{\overline{v}'\}} \text{ (CV-STRUCT)} \qquad \frac{\begin{array}{c}\neg isArray(v) \\ \neg isStruct(v) \\ \neg isLocal(v)\end{array}}{v \mid \sigma \longrightarrow_{v} v} \text{ (CV-OTHER)}$$

5.3.3 Allocation Sinking

Evaluation of primitive operations op_{prim} with constant inputs is trivial because they operate based on primitive numeric values. Evaluation of higher-level object-oriented operations op_{hl} is more difficult because it operates on objects that have an identity that can be observed at runtime.

A critical insight that enables us to optimize those operations away safely is the fact that object allocations do not always *escape*. If an allocation is done locally and used immediately afterward in a way that does not leak its reference to any location outside of the compile-time heap, it can be evaluated away just like any other operation without loss of observable language semantics.

Instead of focusing on the analysis and detection of non-escaping allocations, we provide an evaluation semantics that aims to *sink* the allocations as far down as possible.

We redefine μ to store compile-time heap information about allocations and their current allocation state (`virtual` or `escaped`):

$$\begin{array}{lr}
\mu ::= & \textit{compile-time heap} \\
\quad \emptyset & \textit{empty heap} \\
\quad k \mapsto \texttt{virtual}\ (T,\ \overline{v}), \mu & \textit{virtual allocation} \\
\quad k \mapsto \texttt{escaped}\ l : T, \mu & \textit{escaped allocation} \\
\quad k \mapsto \texttt{var}\ v, \mu & \textit{var allocation}
\end{array}$$

Entries in μ are identified by their key k that serves as a compile-time object identity. Unlike the direct evaluation semantics $\longrightarrow_{eval}$ allocations may also escape, which makes them not eligible for compile-time evaluation.

Virtual entries in μ closely mirror heap state in the direct evaluation semantics. They store an exact type T that models an equivalent of runtime type information and the current state of all fields $\overline{v}$. Fields may refer to other virtual allocations by their key.

Initially, all allocations start as `virtual` heap allocations:

$$\frac{\begin{array}{c} k \notin \mu \quad \overline{T_f} = fieldTypes(n) \\ (\phi, \mu) = \sigma \quad \mu' = k \mapsto \texttt{virtual}\ (n, \overline{\texttt{zero}[T_f]}), \mu \quad \sigma' = (\phi, \mu') \end{array}}{l = \texttt{classalloc}\ n \mid \sigma \longrightarrow_{sink} l = k \mid \sigma'} \text{(SINK-CLASS-ALLOC)}$$

$$\frac{\begin{array}{c} k \notin \mu \quad (\phi, \mu) = \sigma \quad \sigma' = (\phi, \mu') \\ \mu' = k \mapsto \texttt{virtual}\ (\texttt{array}[T], (\texttt{int}\ nv, \overline{\texttt{zero}[T]})), \mu \end{array}}{l = \texttt{arrayalloc}[T]\ \texttt{int}\ nv \mid \sigma \longrightarrow_{sink} l = k \mid \sigma'} \text{(SINK-ARRAY-ALLOC)}$$

Keys k that can only be stored in the compile-time state σ. Whenever a compile-time key is used in the operation op that could not be eliminated, we must *materialize* its allocation right before the use since compile-time identity can not leak into runtime context. We model materialization as one of the last layer evaluation semantics layers $\longrightarrow_{materialize}$ that happens after all other evaluation relations failed to eliminate the operation away (Section 5.3.9).

As long as the allocation did not escape, we can perform all of the operations directly on the compile-time heap state μ similarly to the direct evaluation semantics:

$$\frac{v \mid \sigma \longrightarrow_{v} k \quad k \mapsto \texttt{virtual}\ (T_k, \overline{v}) \in \mu \quad v_s \in \overline{v}}{l = \texttt{fieldload}\ v, s \mid \sigma \longrightarrow_{sink} l = v_s \mid \sigma} \text{(SINK-FIELD-LOAD)}$$

$$\frac{\begin{array}{c} v_1 \mid \sigma \longrightarrow_v k \quad v_2 \mid \sigma \longrightarrow_v v_2' \\ k \mapsto \texttt{virtual}\,(n, \overline{v}) \in \mu \quad v_s \in \overline{v} \\ (\phi, \mu) = \sigma \quad \mu' = k \mapsto \texttt{virtual}\,(T_k, [v_s \mapsto v_2']\overline{v}), \mu \quad \sigma' = (\phi, \mu') \end{array}}{l = \texttt{fieldstore}\ v_1, s, v_2 \mid \sigma \longrightarrow_{sink} l = \texttt{unit} \mid \sigma'} \text{(SINK-FIELD-STORE)}$$

$$\frac{v \mid \sigma \longrightarrow_v k \quad k \mapsto \texttt{virtual}\,(T_k, \overline{v}) \in \mu \quad n_s = resolve(T_k, s)}{l = \texttt{resolvemethod}\ v, s \mid \sigma \longrightarrow_{sink} l = n_s \mid \sigma} \text{(SINK-RESOLVE-METHOD)}$$

$$\frac{v \mid \sigma \longrightarrow_p k \quad k \mapsto \texttt{virtual}\,(T_k, \overline{v}) \in \mu \quad v_0 = isSub(T_k, T)}{l = \texttt{isinstanceof}[T]\ v \mid \sigma \longrightarrow_{sink} l = v_0 \mid \sigma} \text{(SINK-IS-INSTANCE-OF)}$$

$$\frac{v \mid \sigma \longrightarrow_v k \quad k \mapsto \texttt{virtual}\,(T_k, \overline{v}) \in \mu \quad isSub(T_k, T) = \texttt{true}}{l = \texttt{asinstanceof}[T]\ v \mid \sigma \longrightarrow_{sink} l = k \mid \sigma} \text{(SINK-AS-INSTANCE-OF)}$$

$$\frac{v \mid \sigma \longrightarrow_v k \quad k \mapsto \texttt{virtual}\,(\texttt{array}[T], (\texttt{int}\ nv,\ \overline{v})) \in \mu}{l = \texttt{arraylength}\ v \mid \sigma \longrightarrow_{sink} l = \texttt{int}\ nv \mid \sigma} \text{(SINK-ARRAY-LENGTH)}$$

$$\frac{\begin{array}{c} v_1 \mid \sigma \longrightarrow_v k \quad v_2 \mid \sigma \longrightarrow_v \texttt{int}\ nv_1 \\ k \mapsto \texttt{virtual}\,(\texttt{array}[T], (\texttt{int}\ nv_2,\ \overline{v})) \in \mu \\ 0 \le nv_1 < nv_2 \quad v_{nv_1} \in \overline{v} \end{array}}{l = \texttt{arrayload}\ v_1, v_2 \mid \sigma \longrightarrow_{sink} l = v_{nv_1} \mid \sigma} \text{(SINK-ARRAY-LOAD)}$$

$$\frac{\begin{array}{c} v_1 \mid \sigma \longrightarrow_v k \quad v_2 \mid \sigma \longrightarrow_v \texttt{int}\ nv_1 \quad v_3 \mid \sigma \longrightarrow_v v_3' \\ k \mapsto \texttt{virtual}\,(\texttt{array}[T], (\texttt{int}\ nv_2,\ \overline{v})) \in \mu \\ 0 \le nv_1 < nv_2 \quad v_{nv_1} \in \overline{v} \\ (\phi, \mu) = \sigma \quad \mu' = k \mapsto \texttt{virtual}\,(n, [v_{nv_1} \mapsto v_3']\overline{v}), \mu \quad \sigma' = (\phi, \mu') \end{array}}{l = \texttt{arraystore}\ v_1, v_2, v_3 \mid \sigma \longrightarrow_{sink} l = \texttt{unit} \mid \sigma'} \text{(SINK-ARRAY-STORE)}$$

Additionally, we also treat `var` operations a way similar to the object allocations by storing their state in the compile-heap. Due to their second-class nature, they may never escape by construction so all operations on them will always be optimized away:

$$\frac{k \notin \mu \quad (\phi, \mu) = \sigma \quad \mu' = k \mapsto \texttt{var}\ \texttt{zero}[T], \mu \quad \sigma' = (\phi, \mu')}{l = \texttt{var}\ [T] \mid \sigma \longrightarrow_{sink} l = k \mid \sigma'} \text{(SINK-VAR)}$$

$$\frac{v_1 \mid \sigma \longrightarrow_v k \quad (\phi, \mu) = \sigma \quad k \mapsto \texttt{var}\ v_2 \in \mu}{l = \texttt{varload}\ v_1 \mid \sigma \longrightarrow_{sink} l = v_2 \mid \sigma'} \text{(SINK-VAR-LOAD)}$$

$$\frac{\begin{array}{c} v_1 \mid \sigma \longrightarrow_v k \quad v_2 \mid \sigma \longrightarrow_v v_2' \\ (\phi, \mu) = \sigma \quad \mu' = k \mapsto \texttt{var}\ v_2', \mu\sigma' = (\phi, \mu') \end{array}}{l = \texttt{varstore}\ v_1, v_2 \mid \sigma \longrightarrow_{sink} l = \texttt{unit} \mid \sigma'} \text{(SINK-VAR-STORE)}$$

Lastly, if none of the rules above apply, we delegate the evaluation to the constant propagation:

$$\frac{l = op \mid \sigma \longrightarrow_{const} l = op' \mid \sigma'}{l = op \mid \sigma \longrightarrow_{sink} l = op' \mid \sigma'} \quad \text{(SINK-NON-VIRTUAL)}$$

It is important to highlight that allocation sinking is still an optimization even for escaping allocations. For example, fields loads and stores can be performed on the μ state directly, and the memory access operations are elided away. Type-based operations such as method dispatch, casts, or instance checks can also be resolved statically as well. So overall, the more operations we sink a given allocation through, the higher the potential benefit even if the allocation escapes in the end.

Another important insight, is that combination of allocation sinking and constant propagation is sufficient to entirely *partially evaluate* any closed-form programs that consists of op_{prim} and op_{hl} which are the two predominant classes of operations in Scala programs apart from function calls which we are going to cover separately (Section 5.4)

We are going to illustrate how virtual objects are materialized in Section 5.3.9.

5.3.4 Type-based Evaluation

The evaluation rules we have defined so far let us evaluate the high-level operations op_{hl} statically, but they only apply if the allocation remains virtual in μ. Whenever it escapes, we are only left with an opaque type reference $l : T$. Despite the much more limited information, we can still make progress on some of the operations that rely on the type of the value T.

To avoid the loss of information about the escaping allocation type, we extend the base erased type system with reference type qualifiers: `exact` and `nonnull` . Exact references are used to distinguish class references that may only point to the given class but not to any of the subclasses. Non-null references guarantee that the reference is dereferenceable. Qualifiers are not present in the unoptimized intermediate representation and may only appear as a result of type propagation in Interflow. We rely on type qualifiers to aid partial evaluation in the removal of instance checks and virtual calls.

$T ::=$	*types:*
...	
$T_{qual-ref}$	*qualified reference type*
$T_{qual-ref} ::=$	*qualified reference types:*
`exact` T_{ref}	*exact reference type*
`nonnull` T_{ref}	*nonnull reference type*
`exact nonnull` T_{ref}	*exact nonnull reference type*

Escaping allocations start with both qualifiers as part of their escaped opaque type reference. We take advantage of it to evaluate method dispatch and instance checks away:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad isExactRef(T_v) \quad isNonNullRef(T_v) \quad n_s = resolve(T, s)}{l = \texttt{resolvemethod}\ v, s \mid \sigma \longrightarrow_{ty} l = n_s \mid \sigma} \text{ (TY-RESOLVE-METHOD)}$$

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad isExactRef(T_v) \quad isNonNullRef(T_v) \quad v'' = isSub(T_v, T)}{l = \texttt{isinstanceof}\ v, s \mid \sigma \longrightarrow_{ty} l = v'' \mid \sigma} \text{ (TY-IS-INSTANCE-OF-1)}$$

As the last resort we can also use the results of the whole-program analysis to evaluate operations on non-exact references:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad isNonNullRef(T_v) \quad \overline{n} = targets(T, s) \quad |\overline{n}| = 1}{l = \texttt{resolvemethod}\ v, s \mid \sigma \longrightarrow_{ty} l = n_0 \mid \sigma} \text{ (TY-RESOLVE-METHOD-CHA)}$$

Evaluation of instance checks on non-exact references is slightly more involved because of the special treatment of `null` value (which is *not* an instance of any reference type). In case we could not statically ensure that value was not null, we have to emit a runtime null check:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad \neg isExactRef(T_v) \quad isNonNullRef(T_v) \quad \texttt{true} = isSub(T_v, T)}{l = \texttt{isinstanceof}\ v, s \mid \sigma \longrightarrow_{ty} l = \texttt{true} \mid \sigma} \text{ (TY-IS-INSTANCE-OF-2)}$$

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad \neg isExactRef(T_v) \quad \neg isNonNullRef(T_v) \quad \texttt{true} = isSub(T_v, T)}{l = \texttt{isinstanceof}\ v, s \mid \sigma \longrightarrow_{ty} l = \texttt{ine}\ [T]\ v', \texttt{null} \mid \sigma} \text{ (TY-IS-INSTANCE-OF-3)}$$

Casts don't suffer from the same special-case semantics and allow `null` to be cast to any reference type:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad T_v = typeof(v') \quad \texttt{true} = isSub(T_v, T)}{l = \texttt{asinstanceof}\ v, s \mid \sigma \longrightarrow_{ty} l = v' \mid \sigma} \text{ (TY-AS-INSTANCE-OF)}$$

Lastly, if none of the rules applied, we delegate the handling to the underlying $\longrightarrow_{sink}$ evaluation rules:

$$\frac{l = op \mid \sigma \longrightarrow_{sink} l = op' \mid \sigma'}{l = op \mid \sigma \longrightarrow_{ty} l = op' \mid \sigma'} \quad \text{(TY-OTHER)}$$

5.3.5 Canonicalization

Primitive operations with at least one non-constant input can come in multiple shapes. For the sake of normalization, we invert the arguments of commutative operations so that any of the constant arguments are always on the right:

$$\frac{l = op \mid \sigma \longrightarrow_{ty} op_{bin}[T]\, v_1, v_2 \mid \sigma' \quad isCommut(op_{bin}) \quad isConst(v_1)}{l = op \mid \sigma \longrightarrow_{canon} l = op_{bin}[T]\, v_2, v_1 \mid \sigma'} \quad \text{(CN-COMMUT-BIN)}$$

$$\frac{l = op \mid \sigma \longrightarrow_{ty} l = op_{comp}[T]\, v_1, v_2 \mid \sigma' \quad isCommut(op_{comp}) \quad isConst(v_1)}{l = op \mid \sigma \longrightarrow_{canon} l = op_{comp}[T]\, v_2, v_1 \mid \sigma'} \quad \text{(CN-COMMUT-COMP)}$$

$$\frac{l = op \mid \sigma \longrightarrow_{ty} l = op' \mid \sigma' \quad \neg isCommut(op_{comp})}{l = op \mid \sigma \longrightarrow_{canon} l = op' \mid \sigma'} \quad \text{(CN-NON-COMMUT)}$$

This is necessary to simplify the handling of these instructions in the evaluation relations that follow, such as redundancy elimination (Section 5.3.8).

5.3.6 Code Motion

Instructions $\overline{l = op}$ closely follow the order of operations in the original source program. Results of the operations might never be used or may span a long sequence of instructions between the use and the definition.

From the semantics point of view, all pure instructions can be reordered arbitrarily as long as their definition precedes their use. Instructions whose result is never used do not have to be computed, even if we cannot eliminate them through partial evaluation.

We take advantage of the ability to reorder instruction and perform *code motion* that delays instructions as late as possible. This allows us to push down the instructions from the common path into less-frequently-used branches and also perform dead-code elimination if the result of a delayed instruction is never used.

We extend context μ with more additional typed entry that allows us to *delay* instruction materialization:

$$\begin{array}{lll} \mu ::= & & \textit{compile-time heap} \\ \quad \ldots & & \\ \quad k \mapsto \texttt{delayed}\ op, \mu & & \textit{delayed operation} \end{array}$$

So instead of computing, we delay all pure instructions by default. Similarly to virtual allocations, they are going to be materialized whenever their result is used from another non-delayed instruction or escaping allocation:

$$\frac{\begin{array}{c} l = op \mid \sigma \longrightarrow_{canon} l = op' \mid \sigma' \quad isPure(op') \\ (\phi, \mu) = \sigma' \quad k \notin \mu \quad \mu' = k \mapsto \texttt{delayed}\ op', \mu \quad \sigma'' = (\phi, \mu') \end{array}}{l = op \mid \sigma \longrightarrow_{move} l = k \mid \sigma''} \text{(MOV-IS-PURE)}$$

$$\frac{l = op \mid \sigma \longrightarrow_{canon} l = op' \mid \sigma' \quad \neg isPure(op')}{l = op \mid \sigma \longrightarrow_{move} l = op' \mid \sigma'} \text{(MOV-NON-PURE)}$$

Delayed operations reuse the same sinking infrastructure as virtual allocations. Keys that correspond to such operations may appear as the value of local values in ϕ, be the value of any of the fields in virtual objects in μ and additionally allow one to express dependencies between multiple delayed allocations without having to materialize them in the generated code.

As another benefit of code motion expressed above, we also get the benefit of dead code elimination for free. Any delayed instruction, whose result was never used will never be materialized.

The fact that we store both virtual allocations and delayed operations in the same graph allows us to avoid dead code that interleaves allocations and computations, which would be hard to express otherwise. For example, storing the result of an operation to an object field could make it both reachable from both classical liveness and escape analysis points of views. By performing both at the same time, we get more opportunities than in case of those optimizations performed separately.

5.3.7 Combination

Even if a primitive instruction can not be fully evaluated statically, we could still have an opportunity to combine longer chains into smaller ones based on algebraic properties of the underlying operations.

To combine instruction, we first require it to be lifted to the `delayed` graph representation as part of code motion. Delayed instructions form local graph clusters of pure operations that have not been emitted yet. We simplify delayed instructions and produce a modified graph result using $combine$ that relies on domain-specific algebraic invariants about the primitive instructions:

$$\frac{\begin{array}{c} l = op_{bin}[T]\ v_1, v_2 \mid \sigma \longrightarrow_{move} l = k \mid \sigma' \\ (\phi, \mu) = \sigma' \quad k \mapsto \texttt{delayed}\ op' \in \mu \\ (op'', \mu') = combine(op', \mu) \quad \sigma'' = (\phi, \mu') \end{array}}{l = op_{bin}[T]\ v_1, v_2 \mid \sigma \longrightarrow_{combine} l = op'' \mid \sigma''} \text{(COMB-DELAYED)}$$

$$\frac{l = op_{conv}[T]\ v \mid \sigma \longrightarrow_{move} l = op' \mid \sigma'}{l = op_{conv}[T]\ v \mid \sigma \longrightarrow_{combine} l = op' \mid \sigma'} \quad \text{(COMB-OTHER)}$$

5.3.8 Redundancy Elimination

Another opportunity for an elision of operations is redundancy. Any idempotent operation that performed twice can reuse previous statically known result without performing the operation twice.

We extend state σ to contain another component $\rho = \overline{op \mapsto v}$ that maps previously computed operations to their last results. Given the strong normalizing nature of all evaluation rules we have defined so far, we can index the operation into this store to see if it was previously computed:

$$\frac{l = op \mid \sigma \longrightarrow_{combine} l = op' \mid \sigma' \quad (\phi, \mu, \rho) = \sigma' \quad op' \mapsto v \in \rho}{l = op \mid \sigma \longrightarrow_{redundant} l = v \mid \sigma'} \quad \text{(RE-REUSE-REDUNDANT)}$$

$$\frac{\begin{array}{c} l = op \mid \sigma \longrightarrow_{combine} l = op' \mid \sigma' \quad isIdempotent(op') \\ (\phi', \mu', \rho') = \sigma' \quad op' \notin \rho' \quad \rho'' = op' \mapsto l, \rho' \quad \sigma'' = (\phi', \mu', \rho'') \end{array}}{l = op \mid \sigma \longrightarrow_{redundant} l = op' \mid \sigma''} \quad \text{(RE-PERSIST-IDEMPOTENT)}$$

$$\frac{l = op \mid \sigma \longrightarrow_{combine} l = op' \mid \sigma' \quad \neg isIdempotent(op)}{l = op \mid \sigma \longrightarrow_{redundant} l = op' \mid \sigma'} \quad \text{(RE-NON-IDEMPOTENT)}$$

5.3.9 Materialization

As we have seen before, our current evaluation rules might produce keys k as their resulting value for the evaluated operations. Keys are used for delaying operations and allocate objects and variables in compile-time heap μ.

For example, a static assignment $l = op$ can be evaluated to $l = k$. Such assignments are reflected purely in ϕ and never produce any code. On the other hand, if a key k appears as an argument of an actual operation that needs to be performed at runtime, this poses a problem because keys are used as indexes into the compile-time heap state μ and have no runtime semantics.

To avoid the inconsistency, we materialize keys by emitting code. Materialization implies deep traversal of all reachable non-escaping entries in μ starting from the given key k. As the result of traversal, we obtain a subset of all reachable keys that form a potentially cyclic graph of allocations and delayed operations.

We *schedule* the graph into a sequence of operations $\overline{l = op}$ starting from the leftmost argument of a delayed operation and the first field of a virtual allocation. Cycles of virtual

allocations are allocated together before any other operations are emitted.

As the result of scheduling, we obtain new locals for all of the emitted operations and allocations $\overline{l : T}$. We modify μ to replace all of those keys with `escaped` entries that point back to their resulting values.

$$\frac{\begin{array}{c} l = op \mid \sigma \longrightarrow_{redundant} l = op' \mid \sigma' \\ \overline{k} \in op' \quad (\overline{l = op}'', \sigma'') = materialize(l = op', \sigma) \end{array}}{l = op \mid \sigma \longrightarrow_{materialize} \overline{l = op}'' \mid \sigma''} \text{(MAT-OP-HAS-KEYS)}$$

$$\frac{l = op \mid \sigma \longrightarrow_{redundant} l = op' \mid \sigma' \quad \overline{k} \in op' \quad \overline{k} = \emptyset}{l = op \mid \sigma \longrightarrow_{materialize} l = op' \mid \sigma'} \text{(MAT-OP-NO-KEYS)}$$

$$\frac{\begin{array}{c} l = op \mid \sigma \longrightarrow_{redundant} l = v \mid \sigma' \\ (\phi', \mu', \rho') = \sigma' \quad \phi'' = l \mapsto v, \phi' \quad \sigma'' = (\phi'', \mu', \rho') \end{array}}{l = op \mid \sigma \longrightarrow_{materialize} \emptyset \mid \sigma''} \text{(MAT-JUST-VALUE)}$$

5.3.10 Summary

As the result we've obtained a stack of evaluation relations($\longrightarrow_{eval}$, $\longrightarrow_{const}$, $\longrightarrow_{sink}$, $\longrightarrow_{ty}$, $\longrightarrow_{canon}$, $\longrightarrow_{move}$, $\longrightarrow_{combine}$, $\longrightarrow_{redundant}$, $\longrightarrow_{materialize}$). We finalize it with our resulting operation optimizing evaluation $\longrightarrow_{op}$ that transforms a sequence of instructions to their optimized form by invoking materialization on each static assignment in the original program order:

$$\frac{\begin{array}{c} \forall l_i = op_i \in \overline{l = op} : \\ l_i = op_i \mid \sigma_i \longrightarrow_{materialize} \overline{l'_i = op'_i} \mid \sigma'_i \end{array}}{\overline{l = op} \mid \sigma \longrightarrow_{op} \overline{l'_i = op'_i} \mid \sigma'_{last}} \text{(OP-OPT)}$$

We implement formal semantics of $\longrightarrow_{op}$ presented here using a single-pass fused traversal over the sequence of operation that modifies the state similar to an imperative-style interpreter over σ state.

In the best case, after the $\longrightarrow_{op}$, the majority of the visited operations ends up being evaluated in one of the earlier layers and never produces any runtime code. The effects of the code are reflected purely in the resulting σ' which gets propagated across the method control-flow as we are going to illustrate in the next section.

5.4 Intramethod Control-Flow

As we have seen previously, the optimization of static assignments can be expressed as an evaluation rule $\longrightarrow_{op}$. In this section, we are going to generalize this relation to the level of

intramethod control-flow between a sequence of basic blocks $\overline{bb}$ based on the framework of flow-sensitive state σ.

5.4.1 Basic Blocks

Similarly to the handling of static assignments, we can use σ as the flow-dependent context for basic block optimization $\longrightarrow_{bb}$:

$$bb \mid \sigma \longrightarrow_{bb} bb' \mid \overline{\sigma}'$$

A basic block is exposed to multiple incoming states $\overline{\sigma}$ that correspond to incoming control-flow edges. As we are going to illustrate later, it is possible to *merge* those states into a single initial state σ_{start}.

This makes optimization of the whole basic block be a matter of optimization of static assignments within it that are followed by the transformation of the terminator instruction:

$$\frac{\begin{array}{c}\overline{l_{op} = op} \mid \sigma_{start} \longrightarrow_{op} \overline{l'_{param} = op'} \mid \sigma_{end} \\ t \mid \sigma_{end} \mid \sigma_{end} \longrightarrow_{t} \overline{l_t = op_t},\ t' \mid \overline{\sigma}_{out}\end{array}}{l(\overline{l_{param} : T}) :\ \overline{l_{op} = op}\ t \mid \sigma_{start} \longrightarrow_{bb} l :\ \overline{l'_{op} = op'}\ \overline{l_t = op_t}\ t' \mid \overline{\sigma}_{out}}$$

Even though it seems counter-intuitive, we never preserve original basic block parameters by default. Parameters can only appear as an artifact of the state merging that unifies disparate facts about the flow information coming from multiple predecessors. If a block has a single predecessor (which is the common cause due to aggressive partial evaluation), its parameters are always known statically and may only reside in the state σ.

5.4.2 Terminators

Similarly to how we optimize operations, we can eliminate indirect branching using another optimizing relation $\longrightarrow_t$ over terminators:

$$t \mid \sigma \longrightarrow_t \overline{l = op},\ t' \mid \overline{\sigma}'$$

Due to materialization, we might have to emit additional static assignments prior to the terminator instruction. For example, the values within break out terminators $t_{break-out}$ need to be materialized as a result of the terminator evaluation:

$$\frac{\begin{array}{c}v \mid \sigma \longrightarrow_v v' \\ (v'', \overline{l = op}, \sigma') = materialize(v)\end{array}}{\texttt{ret}\ v \mid \sigma \longrightarrow_t \overline{l = op}, \texttt{ret}\ v'' \mid \emptyset} \quad \text{(OT-RET)}$$

$$\frac{v \mid \sigma \longrightarrow_v v' \quad (v'', \overline{l = op}, \sigma') = materialize(v)}{\texttt{throw}\ v \mid \sigma \longrightarrow_t \overline{l = op},\ \texttt{throw}\ v'' \mid \emptyset} \quad \text{(OT-THROW)}$$

$$\texttt{unreachable} \mid \sigma \longrightarrow_t \emptyset,\ \texttt{unreachable} \mid \emptyset \quad \text{(OT-UNREACHABLE)}$$

Handling of jump terminators attempts to always simplify all forms of t_{jump} to its simplest form of a direct unconditional jump. Direct jumps update the state to propagate the parameter values but otherwise do not cause any materialization. This allows us to sink virtual allocations and delayed operations across basic block boundary:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad (\phi, \mu, \rho) = \sigma \quad \phi' = \overline{l_{param} = v'}, \phi \quad \sigma' = (\phi', \mu, \rho)}{\texttt{jump}\ l(\overline{v}) \mid \sigma \longrightarrow_t \emptyset,\ \texttt{jump}\ l \mid \sigma'} \quad \text{(OT-JUMP)}$$

Handling of `if` and `switch` try to simplify them to the direct jump if possible. Otherwise, we have to materialize the underlying value and keep uncertain indirect jump with multiple destinations. Even if the condition or scrutinee of the switch is materialized, it still doesn't require materialization of the passed argument values:

$$\frac{v \mid \sigma \longrightarrow_v \texttt{true} \quad \texttt{jump}\ l_1(\overline{v_1}) \mid \sigma \longrightarrow_t \emptyset, \texttt{jump}\ l_1 \mid \sigma'}{\texttt{if}\ v\ \texttt{then}\ l_1(\overline{v_1})\ \texttt{else}\ l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \emptyset, \texttt{jump}\ l_1 \mid \sigma'} \quad \text{(OT-IF-TRUE)}$$

$$\frac{v \mid \sigma \longrightarrow_v \texttt{false} \quad \texttt{jump}\ l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \emptyset, \texttt{jump}\ l_2 \mid \sigma'}{\texttt{if}\ v\ \texttt{then}\ l_1(\overline{v_1})\ \texttt{else}\ l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \emptyset, \texttt{jump}\ l_2 \mid \sigma'} \quad \text{(OT-IF-FALSE)}$$

$$\frac{\begin{array}{c} v \mid \sigma \longrightarrow_v v' \\ (v'', \overline{l = op}, \sigma') = materialize(v') \\ \texttt{jump}\ l_1(\overline{v_1}) \mid \sigma' \longrightarrow_t \emptyset, \texttt{jump}\ l_1 \mid \sigma_1 \\ \texttt{jump}\ l_2(\overline{v_2}) \mid \sigma' \longrightarrow_t \emptyset, \texttt{jump}\ l_2 \mid \sigma_2 \end{array}}{\texttt{if}\ v\ \texttt{then}\ l_1(\overline{v_1})\ \texttt{else}\ l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \overline{l = op},\ \texttt{if}\ v''\ \texttt{then}\ l_1\ \texttt{else}\ l_2 \mid \{\sigma_1, \sigma_2\}} \quad \text{(OT-IF-UNKNOWN)}$$

$$\frac{\begin{array}{c} v \mid \sigma \longrightarrow_v v' \quad isConst(v) \\ \texttt{case}\ v' \Rightarrow l_i(\overline{v_i}) \in \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})} \\ \texttt{jump}\ l_i(\overline{v_i}) \mid \sigma' \longrightarrow_t \emptyset, \texttt{jump}\ l_i \mid \sigma' \end{array}}{\texttt{switch}\ v\ \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})}\ \texttt{default}\ \Rightarrow l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \emptyset,\ \texttt{jump}\ l_i \mid \sigma'} \quad \text{(OT-SWITCH-CASE)}$$

$$\frac{\begin{array}{c} v \mid \sigma \longrightarrow_v v' \quad isConst(v) \\ \texttt{case}\ v' \Rightarrow l_i(\overline{v_i}) \notin \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})} \\ \texttt{jump}\ l_2(\overline{v_2}) \mid \sigma' \longrightarrow_t \emptyset, \texttt{jump}\ l_2 \mid \sigma' \end{array}}{\texttt{switch}\ v\ \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})}\ \texttt{default} \Rightarrow l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \emptyset,\ \texttt{jump}\ l_2 \mid \sigma'} \text{(OT-SWITCH-DEFAULT)}$$

$$\frac{\begin{array}{c} v \mid \sigma \longrightarrow_v v' \quad \neg isConst(v) \\ (v'', \overline{l = op}, \sigma') = materialize(v') \\ \overline{\texttt{jump}\ l_1(\overline{v_2})} \mid \sigma' \longrightarrow_t \overline{\emptyset, \texttt{jump}\ l_1} \mid \overline{\sigma}_1 \\ \texttt{jump}\ l_2(\overline{v_2}) \mid \sigma' \longrightarrow_t \emptyset, \texttt{jump}\ l_2 \mid \sigma_2 \end{array}}{\begin{array}{c} \texttt{switch}\ v\ \overline{\texttt{case}\ v_0 \Rightarrow l_1(\overline{v_1})}\ \texttt{default} \Rightarrow l_2(\overline{v_2}) \mid \sigma \longrightarrow_t \\ \overline{l = op},\ \texttt{switch}\ v\ \overline{\texttt{case}\ v_0 \Rightarrow l_1}\ \texttt{default} \Rightarrow l_2(\overline{v_2}) \mid \{\overline{\sigma}_1, \sigma_2\} \end{array}} \text{(OT-SWITCH-UNKNOWN)}$$

5.4.3 State Merging

Basic blocks can have multiple incoming edges with completely different incoming states $\overline{\sigma}$ for each of the incoming control-flow edged. To unify potentially disparate flow information, we define a *state merging* procedure.

Whenever multiple states are merged, we produce a single resulting state σ' that contains information that's valid for on all of the potential incoming control-flow paths. Additionally, we might also introduce a sequence of basic block parameters $\overline{l : T}$ to pass any information that differs between states in $\overline{\sigma}$. Original parameters are discarded but can be reintroduced back with more precise type information if necessary.

By definition, state σ is a triple of (ϕ, μ, ρ). Merging of states can be seen as merging of each of the components independently:

1. ρ represents information about previously computed operations results of which can be reused to avoid redundancy. Whenever an entry in ρ is available on all of the incoming paths $\overline{\rho}$ it can be retained in the merged state:

$$op \mapsto v \in \rho \quad \texttt{iff} \quad \forall \rho' \in \overline{\rho} : op \mapsto v \in \rho'$$

   If an entry in ρ is the same for each incoming state, it implies that current state dominates the original definition of op on all paths flowing into the current basic block.

   We do not attempt to persist results with different values v given that operations in ρ are idempotent and can be recomputed cheaply without loss of semantics. If that was not the case, we could have alternatively introduced a new basic block parameter to retain the result on each of the incoming edges (for example if recomputation of a particular operation is known to more expensive than an additional basic block parameter).

2. ϕ represents the current state of all accessible local variables l. Given the semantics of name binding in NIR that we have discussed previously, only locals $\overline{l}$ that are available on all incoming control-flow paths can be accessed:

$$l \in \overline{\phi} \quad \texttt{iff} \quad \forall \phi' \in \overline{\phi} : l \in \phi'$$

   The values of the locals, on the other hand, can be completely different for each of the incoming states $\overline{\phi}$, and unlike entries in ρ we have no easy way to recompute them on demand. So instead, we introduce a new basic block parameter for each of the locals that has two or more distinct values in either of incoming states:

$$l \ \texttt{introduces a parameter iff} \ \ size(\{v \mid l \mapsto v \in \phi', \phi' \in \overline{\phi}\}) \geq 2$$

   The type of each parameter is expressed as the least upper bound of all values on all incoming paths. This way we can propagate reference type qualifiers across the basic boundary without loss of information due to coarse basic block parameter type annotations in the original program.

   Extra care needs to be applied to locals that have key k values and require a new basic block parameter. Those values can not be passed at runtime so we must materialize them on the edge coming from the corresponding predecessor.

3. Compile-time heap storage μ works similarly to the handling of local values $\overline{l}$. We start by computing an intersection of keys $\overline{k}$ available in all of the incoming states:

$$k \in \overline{k} \quad \texttt{iff} \quad \forall \mu' \in \overline{\mu} : k \in \mu'$$

   The resulting state behind any given key is defined as follows:

   (a) If key k escaped in any of the predecessor states, it must be materialized in all the others. The resulting object identity is passed as a new basic block parameter $l : T$ and the resulting key state is marked as escaped. The type T is the qualified type of the escaping allocation that must be the same for all preceding states.

   (b) If key k represents a variable state, it can never escape by construction, so we merge the resulting values of `var` v on each incoming path similar to the values of the locals (i.e., introduce a parameter if there are more then two distinct values).

   (c) If key k is virtual in all predecessor states, we go through each of its fields and merge them similarly to the handling of local values. The resulting state is a new virtual state. In case when field values are identical on all paths, it remains unchanged from the predecessor state.

   (d) If a key k represents a delayed operation, it must be the same on all incoming edges due to the fact that delayed operations always obtain a unique key and are never modified after the initial insertion into μ. We retain operation as delayed in the resulting merged state.

(e) If in the process of handling the resulting keys $\overline{k}$ we transitively reach a key that does not belong to the set, it must be materialized on the corresponding edge.

The state-merging process might trigger the materialization of a key. Given that the process itself relies on whether a given key escaped, we must repeat the merge process whenever a new escaped key is discovered until the escaped key set stabilizes.

5.4.4 Block Processing

1: **procedure** PROCESSBLOCKS($\overline{bb}, \sigma_{initial}$)
2: $queue \leftarrow \{bb_0\}$ ▷ Priority queue of blocks to process.
3: $incoming \leftarrow \{bb_0 \mapsto \{\sigma_{initial}\}\}$ ▷ Maping of incoming states indexed per predecessor.
4: $start \leftarrow \{bb_0 \mapsto \emptyset\}$ ▷ Mapping of merged start states per basic block.
5: $processed \leftarrow \emptyset$ ▷ Mapping to processed blocks and their parameters.
6: **while** $nonEmpty(queue)$ **do**
7: $bb \leftarrow popEarliest(queue)$
8: $queue \leftarrow queue \setminus bb$
9: $\overline{\sigma} \leftarrow incoming(bb)$
10: $(\sigma_{merged}, \overline{l : T}) \leftarrow merge(\overline{\sigma})$
11: **if** $start(bb) \neq \sigma_{merged}$ **then**
12: $invalidate(bb)$
13: $start(bb) = \sigma_{merged}$
14: $bb \mid \sigma_{merged} \longrightarrow_{bb} bb' \mid \overline{\sigma}_{out}$
15: $processed(bb) = (\overline{l : T}, bb')$
16: $incoming = updateOutgoing(incoming, bb', \overline{\sigma}_{out})$
17: **for** $bb_{successor} \in t_{bb'}$ **do**
18: $queue = queue \cup bb_{successor}$
19: **end for**
20: **end if**
21: **end while**
22: $(\overline{bb}', \sigma_{result}) = finalizeProcessing(processed, incoming)$
23: **return** $(\overline{bb}', \sigma_{result})$
24: **end procedure**

Figure 5.2 – Block Processing Algorithm

Now that we have introduced $\longrightarrow_{bb}$, $materialize$ and $merge$, we have enough building blocks to handle a sequence of basic blocks $\overline{bb}$ that represents a method body.

We define an iterative process called *block processing* (Figure 5.3) that handles the state management and block evaluation order.

The process starts with an initial state $\sigma_{initial}$ and a non-empty sequence of basic blocks $\overline{bb}$. Initial state initializes all of the parameters of the entry basic block to an opaque value with the corresponding type. Otherwise, the state is completely blank and contains no additional

information.

Block processing visits one block at a time starting from the entry point bb_0. Entry block is visited in the initial state $\sigma_{initial}$ and produces zero or more successors that are added to the queue. For each of the successors, we also update the $incoming$ map with the corresponding states $\overline{\sigma}_{out}$. The process repeats until no more blocks are left in the $queue$.

In the absence of backward edges, we visit each basic block exactly once. Our work queue prioritizes blocks that appear earlier in the basic block sequence $\overline{bb}$. This implies that every time a block is visited, all of its predecessors have been already processed.

The situation gets a bit more challenging with the addition of backward edges. As we have mentioned previously, well-formed NIR programs can only contain reducible control-flow. This means that we can always partition the edges into forwards and backward sets.

As we have illustrated earlier processing of forwards edges is well behaved and requires no special treatment. Backward edges introduce cycles that must affect the state in which a block is transformed. The first traversal of the block is based purely on incoming forwards edges is insufficient on its own, because it ignores the modified states coming from backward edges.

To address this problem, the process may revisit blocks every time an edge with the new incoming state is discovered. We persist the initial starting state for each visited block in $start$. Whenever a discovery of a new backwards edge modifies the starting state, we rerun the optimization of this block and invalidate all transitive successors blocks that were optimized based on an outdated starting result.

Across the Scala Native's standard library and a set of case study applications, we have observed that most loop bodies are revisited just once. The largest number of invalidations we have seen on our codebase is six invalidations for a method in the standard library that handles integer to string conversion. That code contains multiple nested loops and complicated branching that requires multiple invalidations for the block state to converge. Generally, such coding style is uncommon in Scala which encourages the use of collection combinators instead of explicit C-style loops.

More generally, we do not prove that our algorithm always terminates for all programs. As a precaution, our implementation limits the number of times a single basic block can be revisited to 256 times. In case this limit is reached, a given method is not going to be optimized by Interflow.

5.5 Intermerthod Control-Flow

In the previous section, we have walked through whole-method optimization pass that is guided by the block processing algorithm. All the optimizations that we have discussed so far made judgments only within boundaries of a single method, and completely fail to

handle method calls. To address this problem, we provide two approaches that increase the optimization scope: inlining and method duplication. Additionally, we also define a whole program traversal algorithm and show how it interacts with both techniques.

5.5.1 Inlining

Inlining is one of the most widely used techniques to increase the scope of the optimization horizon. The technique is based on the substitution of the call with an expanded method body.

The primary benefit of inlining is exposure of the method body to the caller context that enables optimizations across the method boundary. This expands the scope of optimizations local to a single method to apply to the mix of caller and callee code. So, for example, we can apply redundancy elimination to remove redundant computation that happened in the caller prior to the call with the same one happening in the callee.

In Interflow, we model inlining as a form of recursive block processing. As we have stated earlier, block processing happens in a particular state $\sigma_{initial}$. In the basic case, we populate this state with opaque type information about every entry block parameter and do not provide any additional information.

Instead of using the coarse initial state, we can take advantage of the precise call-site specific state σ of each individual call and apply block processing to it directly. A naive version of inlining based on block processing can look like:

$$\frac{\begin{array}{c} v_0 \mid \sigma \longrightarrow_v n \quad \overline{v} \mid \sigma \longrightarrow_v \overline{v'} \\ \texttt{def}\ n : T_n = \overline{bb} \quad shallInline(n, \overline{v'}, \sigma) \\ (\overline{bb}', \overline{\sigma'}) = processBlocks(\overline{bb}, [\overline{l_{param} \mapsto v'}]\sigma) \\ (\overline{bb}'', \overline{\sigma''}, l_{result}) = mergeReturns(\overline{bb}', \overline{\sigma'}) \end{array}}{l = \texttt{call}[T]\ v_0(\overline{v}) \mid \sigma \longrightarrow_{naive-inline} \texttt{jump}\ l_{bb''_0}, \overline{bb}'', l_{result} \mid \sigma''}$$

As we are going to see later, we can further improve upon this by tightly integrating inlining into the whole-program traversal.

Given the analogy of block processing to the imperative interpretation, inlining effectively creates an equivalent of nested call frame and then simulates application on the call-site state σ. As a result, we collect the effects of the call reified as the outgoing state σ' with additional code emitted as $\overline{bb}'$ if necessary. In the best case, the whole call is performed completely statically and does not emit any code, but only produces a modified state σ'.

Due to the fact that we want a single resulting output state, we need to merge all return states to a single one using the same merging procedure that we have used previously for processing of regular control-flow merges. Unlike regular processing, we allow returns to contain not-

yet-materialized results. They can remain virtual within the resulting state σ' similarly to the handling of basic block parameters we have discussed earlier.

Inlining heuristic $shallInline$ takes current call-site state and the current values of all arguments to decide if a call should be inlined. It considers the following inlining incentives:

- Method is small. A number of Scala methods do not perform any computation but redirect to another method immediately (e.g., bridge methods).
- Method is a class or module constructor. Constructors often form deep dependency chains due to deep inheritance hierarchy in Scala collections. They typically perform trivial actions such as initializing object fields with given arguments.
- Method is a field getter or setter. Getters and setters abstract away field access to satisfy Scala's uniform access principle but otherwise serve no other purpose.
- Method is explicitly annotated with `@inline`. Interflow respects end-user annotations even if they might produce suboptimal code.
- Method call has arguments that are virtual keys k. Emitting such a call would force materialization incurring additional performance cost compared to an inlined call that can delay or completely avoid materialization of said keys. So we bias inlining decisions to inline calls with such arguments whenever possible.

The presence of an incentive is not sufficient to make an inlining decision. Additionally, we ensure that none of the following conditions that prevent inlining, hold:

- Method is explicitly annotated with `@noinline`.
- Inlining a method overflows the inlining size budget for the current method.
- Inlined method is recursive. We rely on Scala's built-in tailrec support to transform tail recursion into loops.
- All of the arguments are opaque-typed values. As we are going to see later, method duplication is sufficient to improve the context for such calls.

Those limitations limit inlining as it can not be guaranteed to happen at all times. Whenever inlining fails, we have to materialize all call arguments and keep the call operation in the generated code. To avoid loss of information in that case, we perform another technique called method duplication that aims to propagate coarse type information from the caller to the callee.

A cost model could be used to further improve the quality of inlining decisions. In its current form, Interflow does not feature any form of direct performance impact estimation.

5.5.2 Method Duplication and Whole-Program Traversal

```
1: d̄ ← { ... }                                   ▷ Linked program after reachability analysis.
2: queue ← ∅                                      ▷ Queue of methods to visit with given types.
3: context ← ∅                                    ▷ Stack of currently visited or inlined methods.
4: processed ← ∅                                  ▷ Resulting optimized methods.
5:
6: procedure PROCESSPROGRAM(n_entry)
7:     queue ← {(n_entry, signatureTypes(n_entry))} ∪ queue
8:     while nonEmpty(queue) do
9:         (n, T̄) ← pop(queue)
10:        processMethod(n, T̄)
11:    end while
12:    return processed
13: end procedure
14:
15: procedure PROCESSMETHOD(n, T̄)
16:    if visit (n, T̄) ∉ context ∧ (n, T̄) ∉ processed then
17:        push(context, visit (n, T̄))
18:        b̄b̄ = basicBlocks(d_n)
19:        σ_initial = initialState(bb_0, T̄)
20:        (b̄b̄′, σ_result) = processBlocks(b̄b̄, σ_initial)
21:        processed((n, T̄)) = updateDefinition(d_n, T̄, b̄b̄′, σ_result)
22:        pop(context)
23:    end if
24: end procedure
25:
26: procedure REQUESTDUPLICATE(n, T̄)
27:    T̄_visit = visitTypes(n, T̄)
28:    processMethod(n, T̄_visit)
29:    return processed((n, T̄_visit))
30: end procedure
31:
32: procedure VISITTYPES(n, T̄)
33:    if shallDuplicate(n, T̄) then
34:        return T̄
35:    else
36:        return signatureTypes(n)
37:    end if
38: end procedure
```

Figure 5.3 – Program Processing Algorithm

Interflow visits the whole program starting from the application entry point n_{entry} (Figure 5.3). We visit all reachable methods per a sequence of call-site argument types $\overline{T}$ which we refer to as the *duplicate signature*.

Every single pair $(n, \overline{T})$ corresponds to a new *duplicate* of a method d_{dup}. Not all created duplicates survive Interflow since inlining may make some of them unreachable. Signatures do not contain return types; they are inferred through the type propagation within the method body.

Every duplicate goes through the block processing algorithm starting with the coarse initial state $\sigma_{initial}$. As we have alluded previously, the core idea of Interflow is to perform block processing recursively. This means that the current method traversal might trigger an additional nested traversal as long as there are no cycles between the methods we have observed so far. Nested traversals are triggered by $requestDuplicate$ which returns either an optimized version of the requested duplicate or an empty result $\emptyset$ if the request could not be satisfied.

Method duplicates might form cyclic dependencies that prevent their return types from being inferred without doing a fixpoint computation across the cycle. We perform cycle detection by maintaining a stack of currently performed tasks $context$. Tasks could either be `inline` or `visit` task. If a signature is visited a second time (i.e., it is part of a cycle), nested processing fails by returning an original declared method instead of the optimized one to break the cycle.

Now that we have defined the program traversal, we can revisit the handling of calls as an additional evaluation relation $\longrightarrow_{call}$ that extends $\longrightarrow_{op}$ and precedes $\longrightarrow_{materialize}$. The inlining rule we have sketched previously can be rewritten in the following way to take advantage of the program processing state:

$$\frac{\begin{array}{c}\overline{v_0} \mid \sigma \longrightarrow_v n \quad \overline{v} \mid \sigma \longrightarrow_v \overline{v'} \\ d_{dup} = requestDuplicate(n, \overline{T_{v'}}) \\ shallInline(d_{dup}, \overline{v'}, \sigma) \quad \texttt{inline}\ (n, \overline{T_{v'}}) \notin context \\ push(context, \texttt{inline}\ (n, T)) \\ (\overline{bb'}, \overline{\sigma'}) = processBlocks(\overline{bb_{dup}}, [\overline{l_{param} \mapsto v'}]\sigma) \\ pop(context) \\ (bb'', \overline{\sigma''}, l_{result}) = mergeReturns(bb', \overline{\sigma'})\end{array}}{l = \texttt{call}[T]\ v_0(\overline{v}) \mid \sigma \longrightarrow_{call} \texttt{jump}\ l_{bb''_0}, bb'', l_{result} : \mid \sigma''} \quad \text{(CALL-INLINE)}$$

The first difference is that we can take advantage of the stack of currently visited methods $context$ to detect cycles. Unlike most other techniques that take advantage of call graph analysis, we detect cycles *dynamically* as we symbolically execute through the program. This means that a single method can be cyclic or not depending on the context it's being visited in which is generally not easy to express in classic systems.

Moreover, another way we can take advantage of program processing is the ability to first expand the method as a duplicate in the caller context and only then consider it for inlining. This means that we consider already optimized code for inlining, which allows reducing redundant reoptimization of the same code paths and makes it easier to balance between caller and callee size budgets.

If either the inlining heuristic or the cycle detection prevented us from inlining, we persist the call to the duplicate as-is:

$$\frac{\begin{array}{c} v_0 \mid \sigma \longrightarrow_v n \quad \overline{v} \mid \sigma \longrightarrow_v \overline{v'} \\ \neg shallInline(n, \overline{v'}, \sigma) \vee \texttt{inline}\ (n, \overline{T_{v'}}) \in context \\ d_{dup} = requestDuplicate(n, \overline{T_{v'}}) \quad d_{dup} \neq \emptyset \end{array}}{l = \texttt{call}[T]\ v_0(\overline{v}) \mid \sigma \longrightarrow_{call} l = \texttt{call}[T_{dup}]\ n_{dup}(\overline{v'})} \text{ (CALL-USE-DUPLICATE)}$$

This propagates the caller type information across the method boundary and still allows to improve over the original declared method signature. Moreover, the return type of the call is refined by the signature of the duplicate method, which improves precision by enabling type propagation of qualified types back into the caller.

Even without inlining, method duplication needs to be aware of the cycles from the backward type propagation point of view. If the signature we are trying to duplicate is currently on the program processing stack, we bail by calling the original method:

$$\frac{\begin{array}{c} v_0 \mid \sigma \longrightarrow_v n \quad \overline{v} \mid \sigma \longrightarrow_v \overline{v'} \\ \neg shallInline(n, \overline{v'}, \sigma) \vee \texttt{inline}\ (n, \overline{T_{v'}}) \in context \\ requestDuplicate(n, \overline{T_{v'}}) = \emptyset \end{array}}{l = \texttt{call}[T]\ v_0(\overline{v}) \mid \sigma \longrightarrow_{call} l = \texttt{call}[T]\ n(\overline{v'})} \text{ (CALL-NO-DUPLICATE)}$$

5.5.3 Polymorphic Calls

The call handling we have discussed so far works only on the calls that we were able to successfully devirtualize through either allocation sinking or type-based evaluation. However, there are still cases that can not be handled this way, and we need to make sure Interflow can handle them as well.

As a last resort, for methods that have a small number of targets according to the whole program analysis information, we perform an equivalent of polymorphic inline caching that replaces a virtual call with a sequence of static calls guarded by the type test:

$$\frac{\begin{array}{c} v \mid \sigma \longrightarrow_v k \quad k \mapsto \texttt{delayed resolvemethod}\ v', s \in \sigma \\ \overline{n} = targets(typeof(v'), s) \quad |\overline{n}| \leq 4 \\ \overline{l' = op'} = expandPoly(v', s) \end{array}}{l = \texttt{call}[T]\ v_0(\overline{v}) \longrightarrow_{poly} \overline{l' = op'} \mid \sigma'} \text{ (POLY-EXPAND-METHOD)}$$

In the worst case, virtual methods with a large number of method targets are visited conservatively using their original method signature:

$$\frac{v \mid \sigma \longrightarrow_v v' \quad \overline{n} = targets(typeof(v'), s) \quad |\overline{n}| > 4 \qquad \forall n' \in \overline{n} : visitDuplicate(n', signatureTypes(n'))}{\texttt{resolvemethod}\ v, s \mid \sigma \longrightarrow_{poly} \texttt{resolvemethod}\ v', s} \text{ (POLY-METHOD)}$$

This ensures that we optimize all reachable methods, even if we can not reach them statically and take the complete advantage of the inlining and method duplication to optimize across the boundary. The heavy focus on multiple approaches of method devirtualization we have discussed so far are aimed at reducing the chance of this worst-case from happening.

As we are going to see in the next chapter, we can reduce the chance of unoptimized virtual dispatch even further through the use of profiling information.

5.6 Related Work

Prokopec et al. [69] studies the impact of aggressive JIT compilation on Scala collections. Their work postulates that profiling information enables a JIT compiler to better optimize collection operations compared to a static compiler. Based on our performance evaluation (Chapter 7) Interflow obtains comparable results on collections-heavy benchmarks such as Kmeans *purely ahead of time without any profile feedback*. We still observe performance improvements thanks to profile-guided optimization (Chapter 6), but our results suggest that it is not essential to many of our benchmarks.

The Julia programming language [19] performs interprocedural type inference to recover types from dynamically typed programs. Both Julia and Interflow visit programs in graph-free forward dataflow manner [56]. Unlike Julia, Interflow does not rely on fixpoint to recover return types of recursive definitions but instead uses original declared type. Both Julia and Interflow duplicate methods to improve performance. Julia relies on heuristics to prune the number of specialized instances that are possible due to its rich type system. Interflow, on the other hand, uses a simpler type system to avoid this problem altogether. In particular, we do not have any form of generic types apart from arrays, while Julia has tuples, union types, and generic composite types.

Languages with specialized-by-default generics such as C# [46] and C++ [78] rely on the end-user to annotate programs to guide method duplication. Interflow neither relies on generic information (its type system is erased) nor provides any means for the user to influence method duplication. Specialization is automatic and transparent to the end-user. Major C++ implementations such as GCC [77] and Clang [50] expand C++ templates once per use site and later deduplicate generated code at link time. Interflow, similarly to [46], visits each specialized method variant exactly once at link-time, reducing the impact of method duplication on the application compile time.

Scala specialization [36] and Miniboxing [83] extend Scala's erased generics with opt-in annotations to enable code duplication to reduce boxing overhead. Even though they both offer

means to perform method duplication, we do not rely on method duplication to address the boxing problem, but rather as a means to enable devirtualization, which in turn enables a combination of inlining, allocation sinking and partial evaluation to eliminate unnecessary intermediate allocations. Both [36] and [83] create a fixed number of variants per each generic argument, leading to an exponential explosion of the number of specialized variants per generic argument. Interflow only creates duplicates that are reachable through forward dataflow from the application entry point at link time.

Petrashko et al. [66] suggests performing auto-specialization of Scala programs based on context-sensitive call graphs. Interflow does not construct call graphs (or any other analysis artifacts for that matter) but instead performs a single graph-free traversal of the original program in forward dataflow order [56]. We trade off accuracy for compilation speed and intentionally use an erased type system that closely mirrors the runtime type system, rather than Scala's rich type system. Our performance results (Chapter 7) illustrate that Interflow's precision is sufficient to devirtualize typical programs that rely on Scala collections.

Interflow's allocation sinking is based on the work of Stadler et al. [76]. Our allocation sinking directly builds upon the framework of Partial Escape Analysis (PEA) by performing this transformation in the same pass as inlining and partial evaluation. This enables our implementation to perform inlining decisions based on current flow-sensitive state at a given call site. For example, calls with virtual not-yet-materialized arguments are inlined if possible to delay materialization.

LuaJIT [65] and PyPy [27] explored allocation sinking in the context of tracing just-in-time compilers. Their implementation works at the scope of a single execution trace that dramatically simplifies handling of control-flow and offers an equivalent of inlining as the side-effect of trace recording.

Prokopec et al. [68] explored design for an inlining algorithm that relies on inlining trials in combination with a cost model [53]. Their algorithm estimates the impact of inlining decisions using a cost model over high-level intermediate representation that enables inlining based on a cost-benefit analysis. In contrast to their work, Interflow makes decisions purely based on static information. The use of cost modeling for inlining is an exciting direction of the research, but it requires profile information which is not always available in ahead-of-time context.

Similarly to the HotSpot C2 [64] compiler, we perform all key optimizations in a single pass. One of the advantages of our approach compared to C2 is that we use allocation sinking as one of the key inlining incentives. Moreover, we rely on type-driven method duplication as an alternative to inlining to increase the optimization scope.

Partial inlining [59] improves the quality of inling decisions by inlining only the critical part of the method. Similarly to partial inlining, Interflow is able to inline a subset of the called method. We achieve this as a side effect of running inlining and the rest optimizations in the

same context as the optimization of the caller.

5.7 Conclusions

In this chapter, we presented Interflow, a design for an optimizing compiler that performs method duplication, partial evaluation, allocation sinking and inlining in a fused single-pass traversal of the whole program.

As presented here, Interflow is a optimizer that transforms complete programs based on the purely static information. In the next chapter, we are going to explore an extension that augments it with profile knowledge to improve its optimization decisions.

6 Profile-guided Optimization

In the previous chapter, we have walked through a flow-sensitive whole-program optimization called Interflow. At its core, Interflow is a purely static optimization that fuses several well-known techniques into a single graph-free traversal of the whole program.

Even though Interflow is able to perform judgments about the code based on the flow-sensitive information, its weakness lies in the fact that it does not have any means of distinguishing the relative importance of code paths in the program. As a consequence, Interflow optimizes all code equally aggressively to obtain the best runtime performance.

In this chapter, we are going to extend Interflow with support for profile feedback and see how the optimizations we have discussed earlier can be improved based on it. Moreover, we argue for an approach that allows us to use of JIT-style speculative optimizations in the ahead-of-time setting.

6.1 Introduction

Statically-typed, object-oriented programming languages are predominantly implemented using either purely static *ahead-of-time* (AOT) compilation or dynamic speculative *just-in-time* (JIT) compilation.

AOT compilation performs all of the optimizations statically and outputs a precompiled machine-specific binary. This approach produces efficient code for lower-level languages like C that do not impose high abstraction cost that has to be eliminated by the optimizing compiler.

On the other hand, high-level object-oriented programming languages such as Java or Scala are hard to compile efficiently using pure static analysis. Whenever the high-level semantics of the language does not map to the capabilities of the hardware, one has to emulate them with additional runtime logic.

Virtual machines (VMs) bridge the performance gap between high-level programming languages and the underlying hardware through elaborate multi-tier execution pipelines. VMs start their execution using either an interpreter or a simple baseline compiler combined with a dynamic JIT compiler to aggressively optimize hot code paths.

The main advantage of dynamic optimizers is centered around their ability to collect additional information through runtime profiling during earlier tiers of execution. Based on obtained profiles, a JIT compiler can optimize the code by *speculating* that some properties that were observed before are going to always hold in the future. Whenever speculative assumptions are violated, a JIT compiler *deoptimizes* to one of the earlier tiers of execution.

This approach produces a speedup over pure static analysis, but it also introduces a major negative side-effect: *startup cost* due to running unoptimized code during earlier tiers of execution and runtime compilation overhead. These effects cause a negative and subjective perception around VM-based languages as being resource-heavy and slow even though the slowdown is only happening during the initial startup phase and often is diminished once the VM is fully warmed-up.

As an alternative to JIT compilation, we focus purely on AOT compilation. Profile-guided optimization (PGO) is a common technique to augment static compilation pipeline with runtime knowledge:

1. Compile an instrumented version of the program that injects additional code to collect the profile information.

2. Run the instrumented program on a sample workload that represents a typical use of the application.

3. Recompile the program using profile feedback as an additional input to the optimizing compiler.

While this approach has a potential of providing the same detail of profile information to an AOT compiler, it has not been as widely studied and employed as the speculating JIT compilers [11] in the context of high-level object-oriented languages. In particular, the domain of speculative optimization is generally considered to be in the realm of JIT compilers that is not available in the AOT context.

In this chapter, we are going to introduce an approach called white-gray code splitting that can be used to apply arbitrary speculative optimizations without loss of correctness purely ahead-of-time.

6.2 Intuition

One of the key advantages of the just-in-time compilation is the ability to compile code based on speculative assumptions. The overall structure of the speculatively optimized code looks like:

```
if (speculationInvariant()) {
   fastPathImplementation()
} else {
   deoptimize()
}
```

As long as the invariant holds the compiler can emit code that works purely under the invariant condition but might be incorrect otherwise.

In case the invariant check fails, the JIT deoptimizes [64] out of the current compiled code back to the previous tier of execution such as an interpreter or a baseline compiler. Consequently, deoptimization invalidates the previously compiled code.

Whenever deoptimization is triggered, it takes the currently captured runtime state at a deoptimization point and transfers it into an equivalent state of the baseline implementation. After the transfer, the code can resume its execution back in the original baseline implementation with potential for performance degradation. In the meantime, the JIT compiler recompiles the method without relying on the invalidated assumption.

Some of the key optimizations that rely on method invalidation in modern JIT compiler such as HotSpot JVM [49, 64, 71] are the following:

1. **Class Hierarchy Analysis** [33]. In the presence of dynamic class loading, whole-program static analysis requires method invalidation for soundness. If a newly loaded class invalidates the static analysis assumptions used in the already optimized code, it has to be recompiled using the modified class hierarchy information.

2. **Profile-guided Devirtualization** [43]. Virtual call site with a large number of targets can not be optimized statically. Using type profile information, we can distinguish monomorphic and bimorphic call sites that respectively observe one or two types as the result of the profiling information. Such virtual calls can be optimized assuming only those types are ever used at a given location and deoptimize otherwise.

3. **Untaken Branch Pruning**. As long as a code path is never taken based on the profile information, it does not have to be included in the compiled code. Instead, the deoptimization stub is inserted as the replacement in case the branch is visited after the method is compiled.

4. **Error Handling Elimination**. Profile feedback can be used to detect cases when unexpected error conditions such as null dereference have never been observed at a given location. Instead, null dereference can be checked implicitly by relying on the memory fault handler as a means to trigger deoptimization instead of emitting an actual runtime check right away.

Our goal is to find an approach that is equivalent to deoptimization for the ahead-of-time compilation. This way, we can take advantage of the same key optimizations that are performed by modern JIT compilers to improve the quality of the optimized code on the application hot path.

Unlike a JIT compiler, we can not perform any form of dynamic recompilation by design. The optimized code always has to retain a slow path version to preserve the correctness of the compiled program.

A naive approach would include both slow and hot paths in the generated optimized code:

```
if (speculationInvariant()) {
    fastPath()
} else {
    slowPath()
}
```

As originally noted by Paleczny et al. [64], emitting the diamond control-flow is counterproductive as it destroys the precise information acquired through the invariant check.

In Interflow, any diamond control-flow triggers a state merge that computes a new flow state σ that has to be valid for both fast and slow code paths after the merge. Moreover, any of the results will have to be materialized due to the uncertainty of the invariant condition.

The situation is aggravated in combination with aggressive inlining and method duplication that can greatly amplify the code size increase due to duplication of the same hot and slow paths in multiple contexts.

To address the problem, we take the naive version and automatically transform it into an equivalent form that hoists out the slow path into a stand-alone method:

```
if (!speculationInvariant()) {
    return hoistedSlowPath(...)
}
fastPath()
```

We hoist the code for the slow path itself and any of its control-flow successors into a stand-alone version of the method. If the invariant is invalidated, the execution tail calls [45] the

slow path method with arguments that correspond to a snapshot of all locals that are live at that point. In contrast, the deoptimized version may not rely on the invariant and must be compiled conservatively.

Because we terminate early, the slow path will never interfere with the flow state on the hot path. This is possible because the slow path defined this way is not considered to be a predecessor from the intramethod control-flow point of view. As a consequence, the hot path is safe to be optimized assuming the speculative assumptions always hold.

6.3 Collecting Profile Information

Apart from the ability to deoptimize, we also need to obtain profile feedback to find out additional information about the program that can be used for speculative optimization. To collect this information, we run an instrumented version of the program that records method frequency, edge frequency, and type profile feedback.

Instrumentation is performed as a special compilation mode that emits additional code to collect all the relevant runtime events into an in-memory profile data structure.

6.3.1 Profile Information in NIR

We extend NIR with support for profiling data:

1. A new `weight` attribute allows us to store the number of method invocations:

   $$\begin{array}{ll} a ::= & \textit{attributes:} \\ \quad \ldots & \\ \quad \texttt{weight}\ nv & \textit{weight attribute} \end{array}$$

   If a method does not contain weight information, we assume it should be optimized based purely on static knowledge. Methods which were never called must have an explicit zero-weight attribute attached.

2. Indirect jump terminators such as `if` and `switch` may contain additional information about the frequency of each of the outgoing branches:

   $$\begin{array}{ll} t_{jump} ::= & \textit{jump terminators:} \\ \quad \ldots & \\ \quad t_{jump}\ \texttt{weight}\ \overline{nv} & \textit{jump with edge weights} \end{array}$$

   Weight count for each of the branches corresponds to the number of times the individual control-flow edge has been taken. If weight information is not present, we assume that all branches are equally likely. Explicit 0 weights are required to express that some of the branches were never taken.

3. High-level operations that dispatch on the runtime type of the object, such as dynamic method dispatch, may contain a weighted distribution of the receiver types:

$$
\begin{array}{ll}
op_{hl} ::= & \textit{high-level operations:} \\
\quad \ldots & \\
\quad op_{hl}\ \texttt{weight}\ \overline{(T, nv)} & \textit{operation with receiver type weights}
\end{array}
$$

   The weight information is represented as a sequence of pairs of types with corresponding weight, where weight count is the number of times a given receiver-type was observed at this location. The sequence can be empty, which corresponds to instruction that was never executed during the instrumented run. If no weight information is available, we must handle the operation conservatively and only rely on static analysis.

6.3.2 In-memory Profile Representation

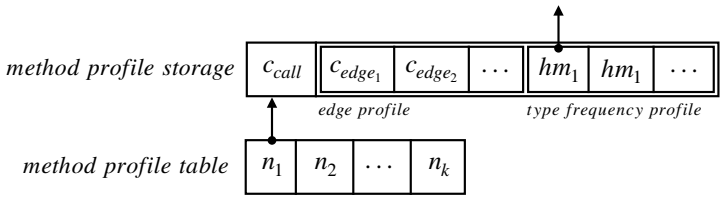


Figure 6.1 – Profile In-memory Represetation.

We collect profiling information using an in-memory data structure (Figure 6.1). The profiling information is split into two levels: a method profile table and a method profile storage for each of the methods.

Each method is identified by a unique numeric identifier that associates it to array slots of the data structure with the profile storage for each of the methods. The storage of this array is allocated statically and contains a single pointer-sized cell for each of the methods that is reachable according to the reachability analysis.

Profile storage represents all the recorded events we've observed for a given method. It's allocated dynamically on the first invocation of the method even though its size is statically known. This lets us avoid allocating profile storage for methods that are never invoked.

The storage is laid out in the following order:

1. At an offset 0, we store the method invocation count. If a method got non-null profile storage, it must have been called at least once, so this count is always non-zero. We use null profile storage as a zero invocation count instead.

2. Invocation count is followed by frequency of all indirect branches (i.e., conditional and switch jumps) within a given method. For each edge (l_{from}, l_{to}) discovered in the

depth-first traversal of the control-flow graph, we assign a single slot that corresponds to the frequency that edge was taken.

3. At the end, we store type profiling information for each virtual dispatch call site. Each call site is identified by a hash map that maps runtime type ids to their corresponding frequency at a given call site. The hash map is allocated lazily upon the first time a given call site is visited.

Each of the counters above is represented as an unsigned 64-bit integer and represents the exact number a given event was observed.

6.3.3 Profile Instrumentation

We instrument the code through a custom lowering $[\![\bullet]\!]_{instrument}$ that builds upon the translation function $[\![\bullet]\!]_{lower}$ we defined for the baseline compilation. In addition to the baseline lowering transformations, we modify the code by injecting calls to the profile collection runtime that operates on the in-memory profile storage model:

1. Instrumentation for the entry basic block to obtain the profile storage pointer and update the call count.

$$
\begin{aligned}
&[\![l(\overline{l_{param} : T}) :]\!]_{instrument} \ \texttt{if}\ isEntryBlock(l) \ = \\
&\quad l(\overline{l_{param} : T}) : \\
&\qquad l_{profile} = \texttt{call}[...]\ @"runtime\_recordcall"(\texttt{long}\ nv_{method-id})
\end{aligned}
$$

   The $runtime\_recordcall$ function is responsible for initializing the profile storage for the given method if it has not been allocated yet and additionally increments the call count. The function returns a pointer to the profile storage for a given method.

2. Instrumentation for each non-direct jump to record the edge frequency.

$$
\begin{aligned}
&[\![\texttt{if}\ v\ \texttt{then}\ l_1(\overline{v_1})\ \texttt{else}\ l_2(\overline{v_2})]\!]_{instrument} \ = \\
&\quad \texttt{if}\ v\ \texttt{then}\ l_1'\ \texttt{else}\ l_2' \\
&\quad l_1' : \\
&\qquad \texttt{call}[...]\ @"runtime\_recordedge"(l_{profile}, \texttt{long}\ nv_{then-edge-id}) \\
&\qquad \texttt{jump}\ l_1(\overline{v_1}) \\
&\quad l_2' : \\
&\qquad \texttt{call}[...]\ @"runtime\_recordedge"(l_{profile}, \texttt{long}\ nv_{else-edge-id}) \\
&\qquad \texttt{jump}\ l_2(\overline{v_2})
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{switch}\ v\ \overline{\texttt{case}\ v_0\ \Rightarrow\ l_1(\overline{v_1})}\ \texttt{default}\ \Rightarrow\ l_2(\overline{v_2})]\!]_{instrument}\ = \\
&\quad \texttt{switch}\ v\ \overline{\texttt{case}\ v_0\ \Rightarrow\ l'_1}\ \texttt{default}\ \Rightarrow\ l'_2 \\
&\quad l'_{11}: \\
&\qquad \texttt{call}[\ldots]\ @"runtime\_recordedge"(l_{profile}, \texttt{long}\ nv_{case1-edge-id}) \\
&\qquad \texttt{jump}\ l_{11}(\overline{v_{11}}) \\
&\quad l'_{12}: \\
&\qquad \ldots \\
&\quad l'_2: \\
&\qquad \texttt{call}[\ldots]\ @"runtime\_recordedge"(l_{profile}, \texttt{long}\ nv_{default-edge-id}) \\
&\qquad \texttt{jump}\ l_2(\overline{v_2})
\end{aligned}
$$

For each of the outgoing branches, we update the branch frequency counter before jumping to the original destination with the given argument values. The edge id directly corresponds to the memory index of the profile storage, so we perform a single increment on a statically known memory offset relative to the $l_{profile}$.

3. Instrumentation for each of the virtual call sites to record the type profile information:

$$
\begin{aligned}
&[\![l = \texttt{resolvemethod}\ v, s]\!]_{instrument}\ = \\
&\quad \texttt{call}[\ldots]\ @"runtime\_recordtype"(l_{profile}, \texttt{long}\ nv_{call-site-id}, v) \\
&\quad l = \texttt{resolvemethod}\ v, s
\end{aligned}
$$

Profile runtime extracts the runtime type id for a given object v and updates the profile storage data structure to reflect the type frequency at a given call site for each of the invocations.

The recorded events are collected throughout the complete application run. Right before termination, we serialize the in-memory representation to disk using a format that mirrors in-memory representation. This data is used in subsequent compilation to provide the profile feedback that is needed to perform speculative optimizations.

6.4 Optimizing based on the Profile Information

Interflow was designed to produce highly optimized code based purely on the flow-sensitive and whole-program static information. In this section, we are going to present a number of extensions that take advantage of profile information to make better optimization decisions.

The profile-guided optimizations rely on a technique we call *white-gray code splitting* that lets us employ similar reasoning to JIT-style speculative optimizations in the ahead-of-time setting.

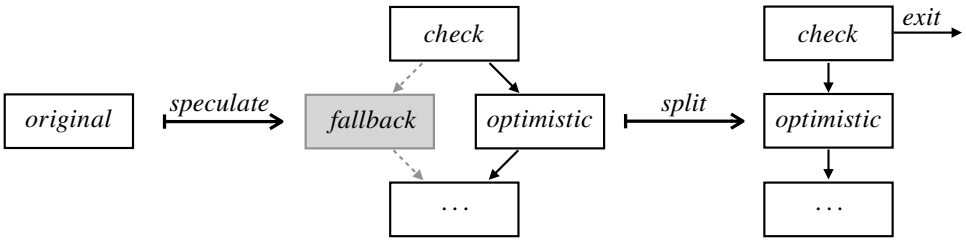


Figure 6.2 – 2-step White-Gray Code Splitting.

6.4.1 White-Gray Code Splitting

As we have outlined earlier, we are interested in performing optimizations that hold purely under a given invariant. To represent such optimizations, we define a framework called white-gray code splitting that consists of two transformation steps:

As a first step, we start with transforming the code into an equivalent naive form of the speculatively optimized code. It represents speculation using an explicit check that leads to an optimistic fast path (white) and falls back to a slow path otherwise (gray).

While this transformation step could work on its own, due to the generated code structure, we can not assume that the invariant holds after we merge back the results from both the optimistic and pessimistic cases.

As a second step, once all of the speculations have been applied, we are going to split the method such that all of the gray code is hoisted out it.

For any method with non-zero weight, the execution always starts with a white basic block. We traverse the method body by following purely the white to white edges in depth-first order. This way, we obtain a transitive closure of all-white blocks reachable from the original method entry point.

Any block that did not make it to the transitive closure of white blocks are marked as gray. White blocks can be demoted to gray if they are only reachable through other gray blocks.

We revisit the transitive closure of white blocks and collect all the edges that go from white to gray. The edge is replaced with a tail call to a hoisted gray method that early returns with the result of the method call. If multiple paths fall back to the same gray block, we reuse the same hoisted method for all of those edges.

Gray methods always start with a gray basic block. Additionally, they must also include any of the blocks that are transitively reachable from it throughout the original method body (including both gray and white blocks). Apart from the original gray basic block parameters,

we must also pass any of the previously defined local variables that are still live in the hoisted code.

Special care needs to be taken for hoisting the gray code out of loops. The hoisted method for such a block would require an effective side-entry into the loop code that does not respect our reducible control-flow restriction. We exempt such blocks from being hoisted and retain them as is in the original method.

As a result of the white-gray code splitting, we obtain an optimized version of the method that contains purely the optimistically optimized code. All of the slow path handling code is hoisted out and may not affect the highly optimized optimistic code path.

We apply white-gray code splitting to all methods that have been invoked at least once throughout the instrumented run. Only a subset of the called methods is eligible for speculation based on the collected profile information. In cases when no speculation has been applied in a first step, we have no code to split out.

The profile-guided version of Interflow takes the transformed code after white-gray code splitting and optimizes white paths exclusively. The simplified control-flow improves the quality of flow-sensitive information. Moreover, slow path hoisting improves the quality of the inlining decisions due to the reduced risk of hitting the code size inlining limit.

To illustrate the framework on a more concrete example, we have implemented two optimizations in this way: profile-guided devirtualization and untaken branch pruning. Both optimizations transform the code through the addition of gray blocks.

6.4.2 Profile-guided Devirtualization

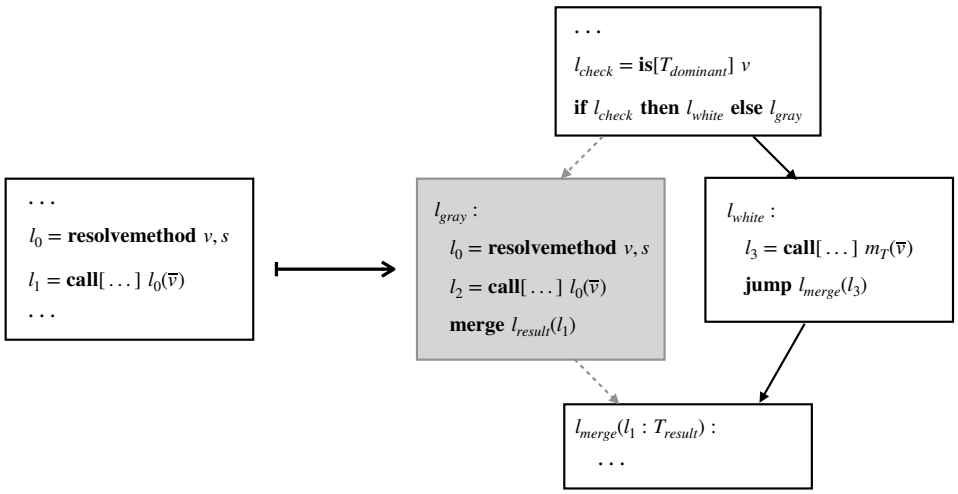


Figure 6.3 – Profile-guided Devirtualization

As we have discussed earlier, Interflow relies on whole-program propagation of qualified

reference types to optimize away virtual dispatch. Nevertheless, polymorphic method calls can not be optimized statically in cases when the call site has more than four potential targets. Type profile feedback gives us another opportunity to revisit this problem.

Given the type profile information for a given call site, one may observe that some call sites have a few dominant types at runtime. Using this information, virtual calls are compiled to a sequence of tests for frequent type cases that redirect control flow to a static call before attempting a virtual method lookup. This opens up the opportunity to inline the implementation of the virtual method for the dominant types.

We implement profile-guided devirtualization using the white-gray block framework introduced earlier. Basic blocks for all of the devirtualized static calls are marked as white, while the full-blow virtual call remains as a gray slow path. The slow path gets hoisted out of the method so only the optimistic case of the call to the implementation for the dominant type remains. The call itself is going to be considered for either inlining or method duplication by Interflow which in turn can enable other optimizations.

Our implementation uses the standard handling of virtual call sites [64] that distinguishes monomorphic and megamorphic call sites. For call sites with two or fewer receiver types, we insert the corresponding type tests and deoptimize if the runtime type does not match any of them. Megamorphic call sites only cache the dominant type that has relatively probability of 0.9 or higher and otherwise fall back to full virtual call as a slow path.

6.4.3 Untaken Branch Pruning

Methods might contain code paths that are rarely or never exercised but are included in the application for correctness (e.g., error handling code). Those code paths contribute to code a considerable size increase that gets amplified if those methods are aggressively inlined.

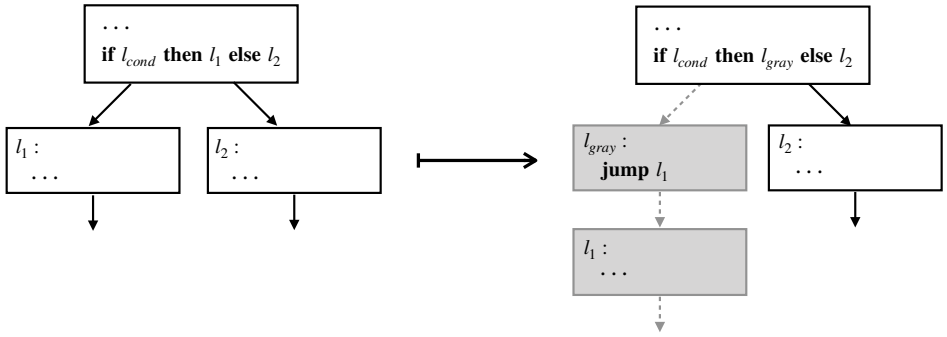
Untaken branch pruning uses edge frequencies collected from the profile feedback to detect the edges that are never taken. For every conditional branch that is never visited, we create an intermediate gray block that immediately forwards to the target of the corresponding branch. This effectively removes the edge from the transitive closure of white to white control-flow graph.

As an example, let us consider a conditional branch that has one taken and one untaken code path. There are two potential effects of this transformation applied to it:

1. If the target of the untaken branch has no other incoming edges, the introduction of an intermediate gray block will mark it and any of its transitive successors as gray:

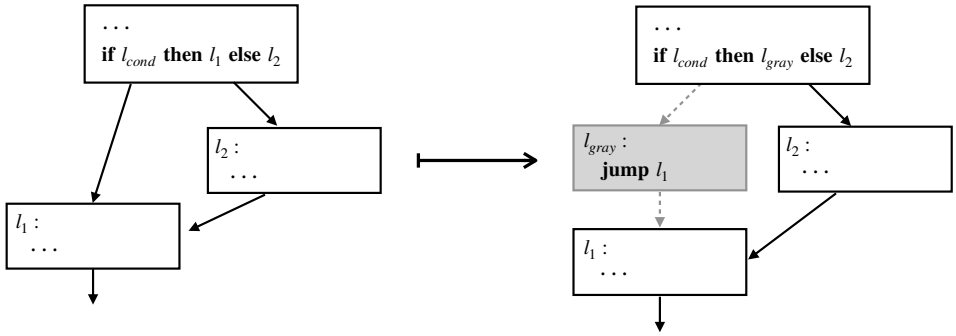
   This can dramatically simplify the control-flow of the method by hoisting out never taken code paths such as error-handling code.

   As a consequence, the extracted slow-path code will never be duplicated if a method is



inlined. Instead, it is going to be compiled exactly once as part of the hoisted out cold path.

2. Otherwise, if the target block has any other incoming edges coming from white blocks, we still get a benefit of the reduced number of predecessors on the white to white control-flow graph:



Since merging of multiple predecessor states is the only way for Interflow to reduce the precision of flow-sensitive information, the decrease of the number of incoming edges has a direct effect on the precision of the resulting merged state.

In the very best case, through the introduction of the gray blocks, we can reduce the number of incoming edges from multiple to a single edge. This means that we do not need to perform any state merging and can keep the predecessor state as is without any loss of information.

As a consequence of untaken branch pruning used in combination with white-gray code splitting, we are left with white to white control-flow where aggressive optimizations have the potential to make the most impact. In the very best case, we can go from a complicated control-flow in the original program to effectively linear code with side exits hoisted out of the hot path.

6.4.4 Profile-guided Inlining and Method Duplication

Interflow employs inlining and method duplications as a means to increase the optimization context beyond the scope of a single method. Both techniques are prone to code size increase due to the fact that they duplicate the same code in multiple contexts.

Profile feedback gives us another angle to reconsider when those optimizations should apply. Based on the relative method frequency we distinguish the treatment of *cold* and *hot* methods:

- Cold methods are the methods that are never called throughout the instrumented run. Despite the best efforts to minimize the number of reachable methods through reachability analysis and flow-sensitive optimization, not all methods can be excluded statically.

  We avoid both inlining and duplication of the cold methods. Calls to such methods may only appear on the gray code paths of another method or be called from another cold method. Inlining and method duplication of the cold paths is unlikely to increase the runtime performance and risks contributing to the binary size increase.

- Hot methods are the methods that contribute top 80% of all instrumented calls (which tends to correspond to less than 10% of the called methods on our case benchmarks). These methods are the major contributors to the runtime performance and should always be optimized aggressively.

  We incentivize the inlining to inline them as long as they do not violate any of the original inlining restrictions. If inlining fails, hot methods are duplicated based on the call site type information instead.

If a method is neither cold nor hot, we optimize using the same optimization heuristics we have discussed for purely static optimization.

6.4.5 Optimizing Cold and Hoisted Methods

Cold and hoisted methods created after white-gray code splitting must remain in the generated code to preserve the correctness of the program after the profile-guided optimization. The profiling information of an instrumented run, might not reflect the exact behavior of all of the subsequent application runs and the cold paths might still be taken on the some of the executions.

From the optimization point of view, we assume that those methods are never going to be called and optimize them for the best size rather than the best runtime performance:

- **Method traversal**. Interflow's method traversal is based on the propagation of the flow-sensitive context σ. Interflow aggressively tries to sink allocations and pure operations by emitting them the first time they are used.

This can lead to code size increase due to the fact that a single operation or allocation sunk down through a branch that could not be eliminated might duplicate the same code multiple times in some of its successors.

For cold methods, we can avoid the risk of code size increased by restricting the context propagation not to perform any code motion. Allocations are always emitted, and instructions are preserved in their original order.

- **Inlining**. Inlining into the cold method might still decrease the overall method size through the removal of small methods such as field accessors, bridges, and constructors which are often trivial and can be eliminated from the program if we inline them into all of their callers.

  We do not consider aggressive inlining incentives such as inlining of hot methods or inlining based on the flow-sensitive state to avoid the potential code size increase of the cold method.

- **Method Duplication**. Given the reduced inlining incentives, more of the call sites become eligible for method duplication. Similarly to inlining, method duplication is a technique that increases code size by specializing a method to a sequence of more precise types propagated from the caller.

  Propagating the type information from the cold path can contribute to a significant impact on the binary size. As a consequence, we prohibit duplication initiated from a cold method.

6.4.6 Profile-guided Native Code Generation

LLVM [51] provides built-in support for profile metadata that can improve the code quality of the generated machine code. We annotate emitted code with method call and branch frequency information collected through profile instrumentation.

The additional profile metadata allows LLVM to optimize the code layout of the hot paths by positioning the more likely branches together in the generated machine code. Besides, the branch weight metadata provides additional feedback to the loop optimizer about the relative importance and frequency of the loop edges and trip count.

Additionally, we annotate the cold and hoisted method with a directive that forces them to be optimized for size rather the best runtime performance. This is equivalent to compiling the lowered code of those methods using `-Os` optimization pipeline.

Similarly to our optimizer, this lets LLVM to avoid optimizations that have a risk of increasing code size and use instruction selection algorithm with the smallest size output rather than best peak performance. Moreover, it also enables the use of optimizations that aim primarily to reduce code such as outlining that are not used for regular compilation.

6.5 Related Work

Speculative feedback-driven optimizations have been researched in-depth in a number of commercial and open-source JVMs [64, 71, 86]. We implement profile-directed inlining, untaken branch pruning, and polymorphic inline caching similarly to well-known JIT compilers for Java.

The main difference in our approach is that we do not perform any compilation at runtime, but instead rely on white-gray code splitting to separate optimistic hot path from the pessimistic gray path that gets hoisted out into stand-alone methods.

Arnold et al. [12] performs an in-depth survey of techniques used in high-performance JVM implementation. One of the insights of their work is the discussion of the connection between speculative optimization and multi-versioning in static compilers. White-gray code splitting can be seen as a direct bridge between the two approaches that does not rely on runtime recompilation to break the diamond control-flow as we discussed earlier.

Deoptimization to optimized code has been previously explored by Wimmer et al. [84]. We do not attempt to keep a single copy of a method for all deoptimization points but instead generate a new method explicitly tailored for each deoptimization location. Most of the time, those methods are rather small as they only contain a small subset split from the original method.

Combination of white-gray code splitting and speculative untaken path pruning produces the same effect as procedure splitting [67]. Rather than relying on procedure splitting as purely a code layout and memory locality optimization, we advocate for compiler design that relies on code splitting to emit both slow path (gray) and fast path (white) code at the same and can reliably optimize assuming only the white path is taken. Our implementation uses tail calls to stand-alone methods rather than long jumps to transfer control-flow to split procedures. This implementation is mostly dictated by the fact that we emit LLVM IR rather than machine code as the result of our compilation.

LLVM [51] supports profile-guided block placement [67] based on information collected through code instrumentation. We built directly upon this work by relying on the same implementation.

A number of more techniques have been developed to decrease the runtime cost of profiling instrumentation. Ball and Larus [15] suggests taking advantage of the static control-flow structure to decrease the number of profiling counters in instrumented versions of the code. Path profiling can provide the same information as edge profiling with reduced instrumentation cost.

Duesterwald and Bala [40] perform hot path prediction based on Next Executing Tail (NET) model. Their approach maintains frequency counts purely for backward taken edges of the loops. Whenever counter reaches a certain threshold, they record an execution trace through

the loop body. Their approach manages to provide a reasonable estimate of hot paths with extremely low overhead.

Tracing JIT compilers [14, 27, 65] rely on trace profiling to extract hot paths out of the programs. The hot paths are optimized aggressively and contain side exits to bail out in case some of the trace conditions have been invalidated. White-gray code splitting coupled with untaken branch pruning achieves a similar effect by isolating the optimistic fast path execution and bailing out in case any of the optimistic invariants have been invalidated.

Nuzman et al. [60] explored the application of dynamic recompilation to statically compiled languages such as C++. Their approach relies on an existing Java JIT compiler repurposed to compile C/C++ programs. They compile programs as fat binaries that include profiling instrumentation and recompile the application code at runtime once the profiling information has been collected. Compared to them, our approach relies on a separate run of an instrumented code to collect profile information and doesn't perform any runtime recompilation in the final optimized code.

6.6 Conclusion

In this chapter, we introduced a technique, called white-gray code splitting that enables the use of speculative optimizations in the ahead-of-time setting.

We have implemented profile directed devirtualization and untaken branch splitting as two case study optimizations build on top of white-gray code splitting framework. Besides, we have also augmented our previous purely static optimizations such as inlining and method duplication, with profile knowledge.

The combination of these techniques produces efficient code that does not depend on multi-tier compilation to achieve high performance and gets the full advantage of the zero-overhead startup time that is inherent to AOT-compiled code.

In the following chapter, we are going to perform a performance evaluation that contrasts our baseline compilation, flow-sensitive static optimization, and profile-guided speculative optimization implemented as part of the Scala Native project.

7 Performance Evaluation

In this chapter, we report performance results of our implementation and compare it against ahead-of-time compiled code produced by Graal Native Image and warmed-up just-in-time compiled code of the HotSpot JVM.

Specifically, we are interested in evaluating:

- Performance overhead of our garbage collection techniques.
- Impact of our optimizations on peak performance.
- Impact of our optimizations on binary size.
- Warm-up profile relative to just-in-time compiled code.
- Peak performance relative to just-in-time compiled code.

As the result of this evaluation, we aim to evaluate if the techniques used in Scala Native are sufficient to match the peak performance of just-in-time compiled code of the HotSpot JVM and quantify the impact of those techniques.

7.1 Environment

We use the following versions of the software projects for the evaluation:

- A development snapshot of Scala Native 0.4.0 in combination with LLVM 8.0.
- OpenJDK 1.8.0_212 which refer to as HotSpot JVM.
- Graal Native Image 19.1.0 (EE) which we refer to as Native Image.
- Ubuntu 18.04.2 LTS.

Benchmarks are executed on a workstation-class machine with the following hardware:

- Intel i9 7900X CPU locked at a fixed 4GHz frequency.
- 128GB of DDR4 3200 MHz memory.
- 512GB of Samsung SSD storage connected over NVMe.

The machines uses a clean install of the minimal distribution of Ubuntu Server [5] and is used purely for the performance evaluation.

7.2 Configurations

We compare performance of the following runtime configurations:

1. **Baseline Compilation** is the simplest compilation mode supported by Scala Native that relies on whole-program analysis for devirtualization. Using the baseline compiled code, we compare our currently supported memory management strategies relative to the No GC collector.

2. **Interflow** is our main static optimizer. It performs a flow-sensitive optimization of the whole program in a single graph-free traversal of the whole program.

   In addition to the full-fledged Interflow, we also evaluate a conservative variant of the same optimizer that does not perform method duplication. Additionally, it also does not consider virtual not-yet-materialized arguments as an inlining incentive.

3. **Interflow with PGO** makes decisions taking both static knowledge and profile data into account while optimizing the program.

4. **Graal Native Image** [2] is a state-of-the-art AOT compiler for the JVM-based languages. Similarly to Scala Native, it relies on whole program analysis and profile data to emit highly optimized statically linked binaries.

5. **HotSpot JDK** is the reference JDK implementation. Its C2 [64] JIT compiler is conventionally used as the baseline for performance evaluation in the JVM ecosystem [9, 41, 71, 86].

7.3 Benchmarks

Our performance evaluation relies on Scala Native benchmarking suite [74]. The majority of benchmarks (bounce, json, cd, deltablue, richards, permute, nbody, mandelbrot) are based on the original benchmarks of Marr et al. [55]. Scala Native's version of these benchmarks uses Scala collections to replace Java-style looping constructs. The Permute benchmark uses the built-in permutations method of Scala Collections, rather than its own implementation to compute permutations.

Kmeans and Sudoku are exclusively collection benchmarks. Kmeans is the same benchmark as the one studied by Prokopec et al. [69].

Tracer benchmark is a raytracer written idiomatic Scala using an object-oriented representation for the Scene that includes the use of Scala collection's lists, ranges, and iteration over them.

Brainfuck benchmark is a naive interpreter of the Brainfuck [58] esoteric programming language. It exercises recursive descent parsing and interpretation based on pattern matching

of a hierarchy of case classes. The interpreter runs a program that generates the lyrics of a programming folklore song 99 Bottles of Beer [48].

Gcbench is a binary trees benchmark by Boehm and Weiser [26] that is commonly used to evaluate the throughput of the garbage collectors.

Rsc [29] is a command-line tool that computes signatures information for Scala programs. It parses Scala source code into an abstract syntax tree and constructs a graph of semantic information that describes it. Our benchmark uses Rsc to compute signatures for a regular expression engine based on RE2 [32].

7.4 Methodology

For each of the benchmarks, we record 4000 in-process iterations over 20 independent runs. We record all measurements without discarding any of the data early on.

Barrett et al. [17] suggests that at least 2000 in-process iterations are required to evaluate VM-based language implementations. We audited the performance results of all of the benchmarks (Appendix A) running on the HotSpot JVM and concluded that our benchmarks require up to 2500 iterations to warm-up. AOT-based implementations warm-up much faster.

Our warm-up charts show a single sample execution run that contains individual data points against the current in-process iteration number for the first 300 iterations. The data points correspond to the pure raw data as recorded without any post-processing.

Our warmed-up performance charts show box plots of the running time of the last 1000 of 4000 iterations across the 20 runs (less is better). Box plots use quartiles for the box itself and first and 99th percentile for the whisker lines. Any data points outside of this range are displayed with individual dots outside the whisker ticks.

We generally do not discard outliers and keep them as explicit outlier dots on the box plots. The only exception is No GC configuration. Given that we are only interested in looking at it as a case of zero-overhead garbage collection, we discard results, which are over ten times slower than the median. These results correspond to calls that memory map the next segment of 4GB memory used for allocation (and there are no more than 32 of such data points across a single run based on the amount of total memory on the workstation machine).

Another difference in the No GC configuration is that since it does not perform any garbage collection, it exhausts the complete 128Gb of the available memory before the end of a single 4000 iteration run on some of the benchmarks. We adjust the number of in-process iterations for cd (1000), kmeans (1000), gcbench (200), kmeans (2000), nbody (2000), rsc (2000). In those cases, we discard half of the measurements to account for warm-up.

For the ahead-of-time compilation configurations, we also have a look at the binary size

emitted for each benchmark. No additional post-processing steps have been performed on the binary (such as symbol stripping), and it corresponds to the size of the direct output of the corresponding compiler.

gc	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
none	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
commix	1.00	0.99	1.05	0.99	1.05	0.97	0.98	0.98	0.94	1.01	1.00	1.01	1.09	1.00	1.01
immix	1.02	1.03	2.33	1.14	1.60	1.71	1.27	1.35	1.29	1.27	1.05	1.25	2.21	1.01	1.34
boehm	1.03	1.06	8.69	2.23	6.05	3.00	2.33	1.81	1.98	1.31	1.42	1.79	4.46	1.01	2.16

Figure 7.1 – Baseline running time at 50 percentile, normalized by No GC, less is better.

7.5 Baseline Compilation and Garbage Collection

As the first comparison in our evaluation we consider performance of our garbage collection strategies under the baseline compilation model (Figures 7.1, 7.2).

No GC provides a baseline to compare the other collectors against. Given that we discard the measurements that involve memory mapping overhead, it presents a case of a garbage collection strategy that does not incur any runtime overhead.

Fully conservative Boehm GC generally can not keep up with No GC on most of our benchmarks. Most notably, the 50 percentile of the running time of a single iteration on permute benchmark is over 8 times slower, while tracer gets up to 6 times slower, gcbench is over 4 times slower. Benchmarks such as mandelbrot, nbody, kmeans, cd, sudoku are relatively less affected but are still consistently slower than the baseline. Across the whole benchmark suite, the geometric mean suggests a 2.16 times slower performance when baseline compilation is coupled with Boehm GC rather than the No GC.

Our initial implementation of Immix GC offers a significant performance uplift compared to Boehm. The majority of our workloads fall within 30% overhead compared to the baseline. Geometric mean indicated 1.34 slow-down compared to the No GC.

Finally, Commix GC shows our current best-of-breed performance. It is able to get extremely close to the performance of the baseline compiled code in combination with No GC. Across the benchmark suite, we see just a 1% degradation compared to our target performance. In fact, some of the benchmarks show a slightly better runtime performance while running with a garbage collector.

Given that Commix is our best garbage collector, we are going to use it for the remaining performance comparisons.

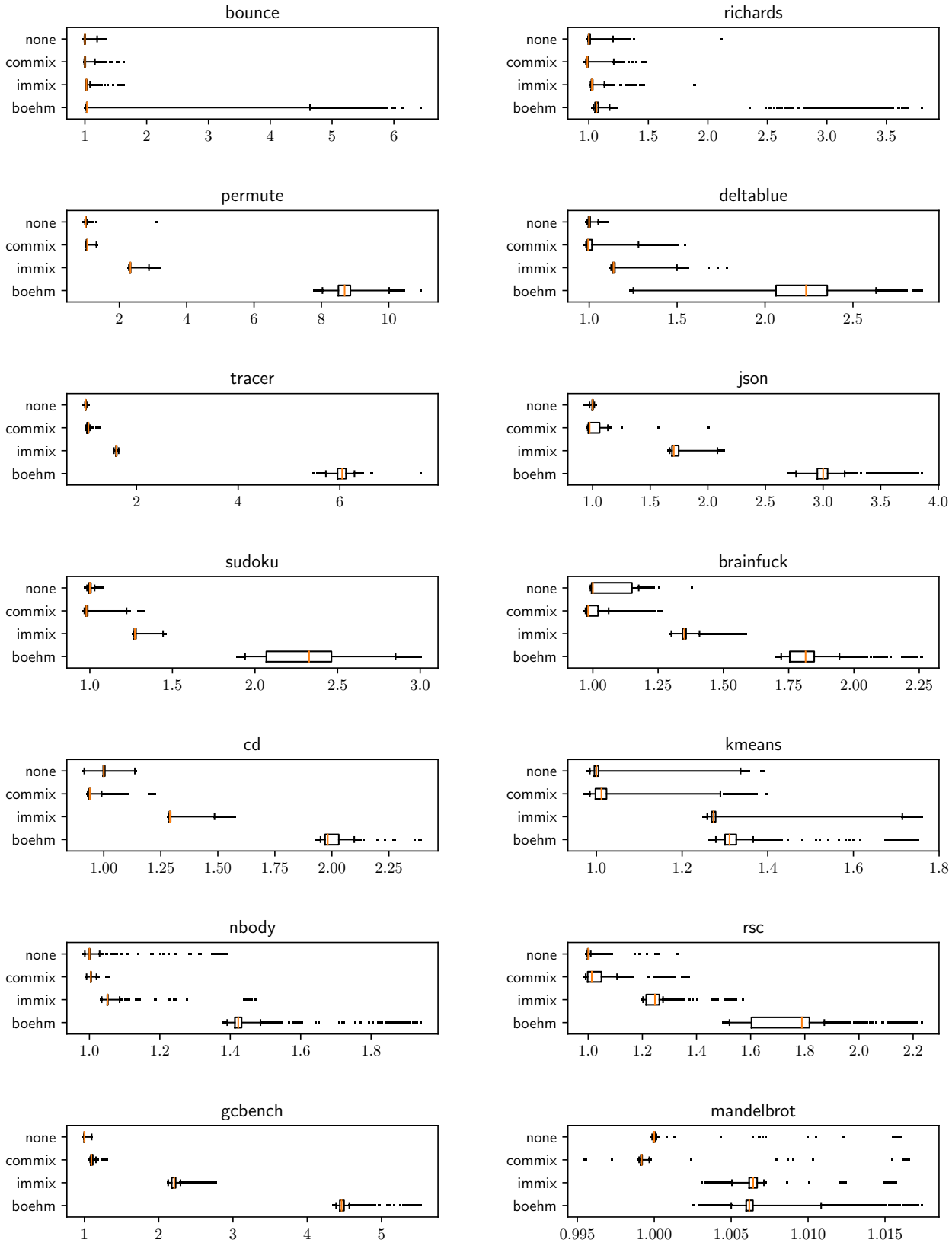


Figure 7.2 – Baseline compilation performance, less is better.

mode	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
pgo	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
aggressive	1.14	1.09	1.13	1.04	1.11	1.08	1.20	1.60	1.17	1.11	1.00	1.24	0.98	1.00	1.13
conservative	2.21	1.12	1.42	1.39	1.40	2.47	1.37	3.05	1.36	2.18	1.30	1.51	1.02	1.00	1.54
baseline	2.66	1.27	8.63	2.39	2.10	4.83	2.18	4.27	1.49	2.34	1.46	2.06	1.18	0.99	2.24

Figure 7.3 – Interflow running time at 50 percentile, normalized by Interflow with PGO, less is better.

mode	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
conservative	0.63	0.67	0.67	0.68	0.67	0.67	0.68	0.69	0.66	0.67	0.60	0.74	0.67	0.67	0.67
aggressive	0.70	0.79	0.81	0.82	0.82	0.86	0.97	0.90	0.82	0.82	0.69	4.56	0.80	0.80	0.92
pgo	0.64	0.71	0.72	0.69	0.71	0.76	0.73	0.61	0.77	0.70	0.63	0.94	0.73	0.74	0.72

Figure 7.4 – Interflow binary size, normalized by baseline compilation, less is better

.

7.6 Flow-sensitive and Profile-Guided Optimization

We evaluate the impact of flow-sensitive and profile-guided optimization relative to the performance of the baseline compiled code we have explored earlier (Figures 7.3, 7.4, 7.5)

As an intermediate step between fully aggressive optimization and baseline compilation, we also consider a conservative variant of Interflow that does not perform method duplication and takes more conservative inlining decisions. Conservative mode offers the best binary size (33% geomean improvement over baseline) due to the fact that it only performs a limited amount of inlining.

Aggressive optimization gets us the best runtime performance in the absence of profiling information. In particular, we see near 1.98x geomean improvement compared to the baseline-compiled code. The increased performance comes at the cost of increased code size relative to the conservatively optimized code, but it still close to the size of the baseline compiled code except for Rsc benchmark.

The Rsc benchmark is an example of the worst-case impact of the method duplication in the absence of profiling information. The benchmark uses a large hierarchy of classes that represents abstract syntax trees. Method duplication took the opportunity to specialize some of the methods for concrete tree subtypes at the expense of significant code size increase. These results suggest that we may have overfitted method duplication for the best runtime performance without enough concern for potential binary size increase.

Profiling information eliminates this issue since we make inlining and duplication decisions based on the weights attributed to methods. Moreover, when coupled with white-gray code splitting, we are able to enjoy the best runtime performance (2.24x faster than baseline) at a modest binary size increase (7.5% larger than the conservative mode).

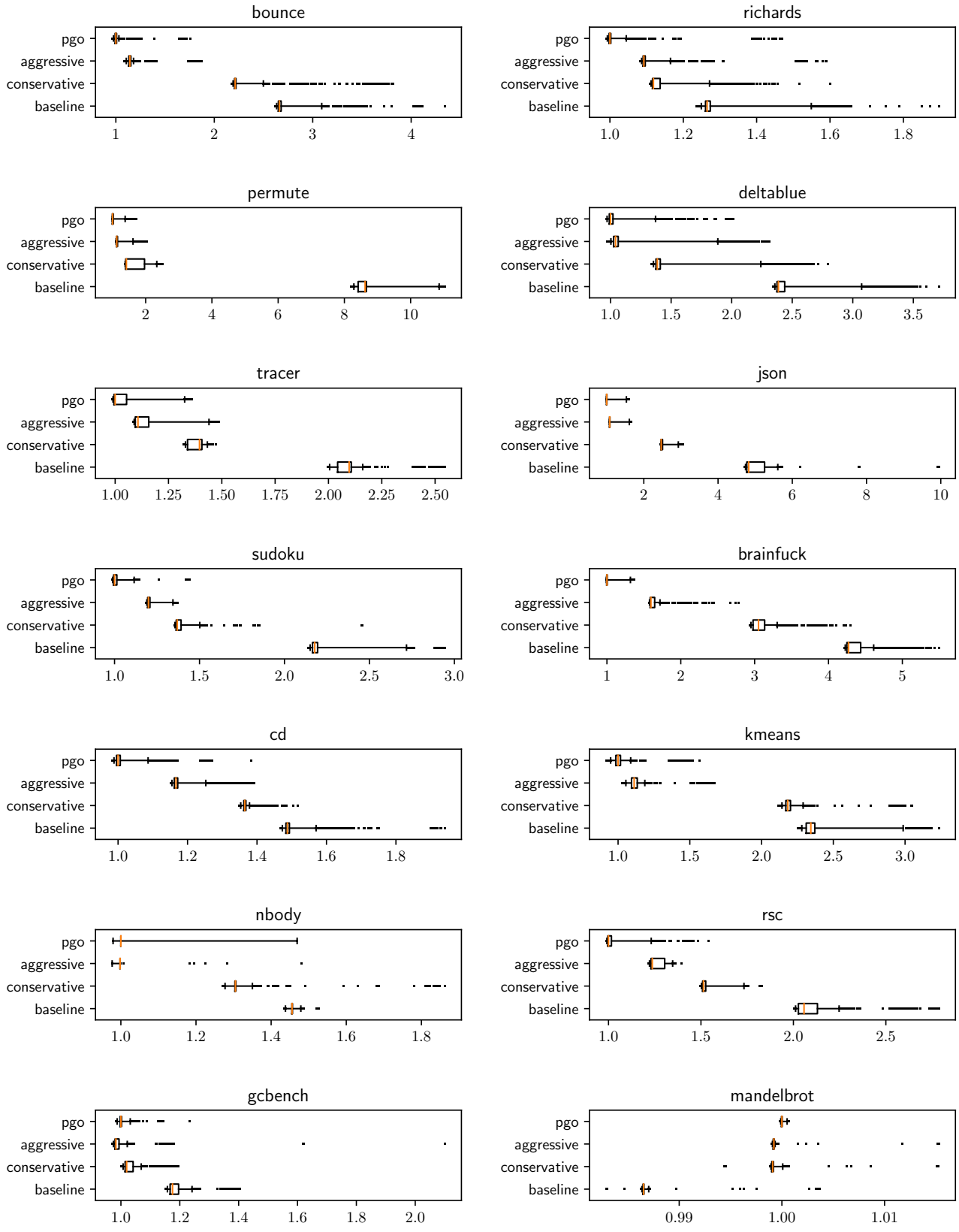


Figure 7.5 – Performance impact of flow-sensitive and profile-guided optimization, less is better.

	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
interflow+pgo	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
interflow	1.14	1.09	1.13	1.04	1.11	1.08	1.20	1.60	1.17	1.11	1.00	1.24	0.98	1.00	1.13
nativeimage+pgo	1.03	1.09	0.93	1.45	1.85	2.49	1.49	2.87	1.58	1.73	1.24	2.07	1.66	1.11	1.53
nativeimage	4.08	1.49	2.64	3.50	1.99	3.68	2.28	5.55	1.97	3.96	1.83	2.24	1.85	1.15	2.49

Figure 7.6 – Native Image running time at 50 percentile, normalized by Interflow with PGO, less is better.

	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
interflow+pgo	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
interflow	1.10	1.13	1.12	1.19	1.17	1.13	1.33	1.47	1.08	1.17	1.10	4.83	1.10	1.09	1.29
nativeimage+pgo	3.63	3.62	3.66	3.90	3.61	3.50	3.66	3.67	3.69	3.74	3.63	1.21	3.62	3.50	3.37
nativeimage	3.47	3.47	3.35	3.55	3.42	3.21	3.14	3.37	3.04	3.35	3.43	1.16	3.38	3.35	3.10

Figure 7.7 – Native Image binary size, normalized by Interflow with PGO, less is better.

7.7 Performance relative to Native Image

Native Image [2] is a state-of-the art ahead-of-time compiler that takes advantage of the Graal [86] optimizing compiler to emit whole-program optimized code. It offers support for both purely static and profile-guided optimization. We compare both our flow-sensitive and profile-guided results with Native Image (Figures 7.6, 7.7, 7.8).

Interflow is able to surpass the performance results by the static variant of Native Image optimized code. Moreover, we produce smaller binaries on most benchmarks apart from the rsc, that we have discussed earlier.

With the addition of profile-guided optimization, Interflow can outperform Native Image by a factor of 1.53x across our benchmark suite. Moreover, we generate consistently smaller binaries ranging from 1.2 to 3.69 times smaller on these benchmarks.

The performance gap with Native Image gets bigger on benchmarks such as gcbench, which is sensitive to the garbage collector performance. Gcbench sees a 1.66 performance degradation compared to Interflow results. This benchmark focuses purely on a synthetic high allocation rate to stress out the garbage collector.

In addition to binary size and warmed-up performance, we also study individual run charts to gain an insight into the warm-up behavior (Figure 7.9). In case of Scala Native, the warm-up period is caused by the automatic scaling of the garbage collected heap and may lead to performance degradation until the heap size stabilizes as seen on permute, deltablue, tracer,

brainfuck, and nbody benchmarks.

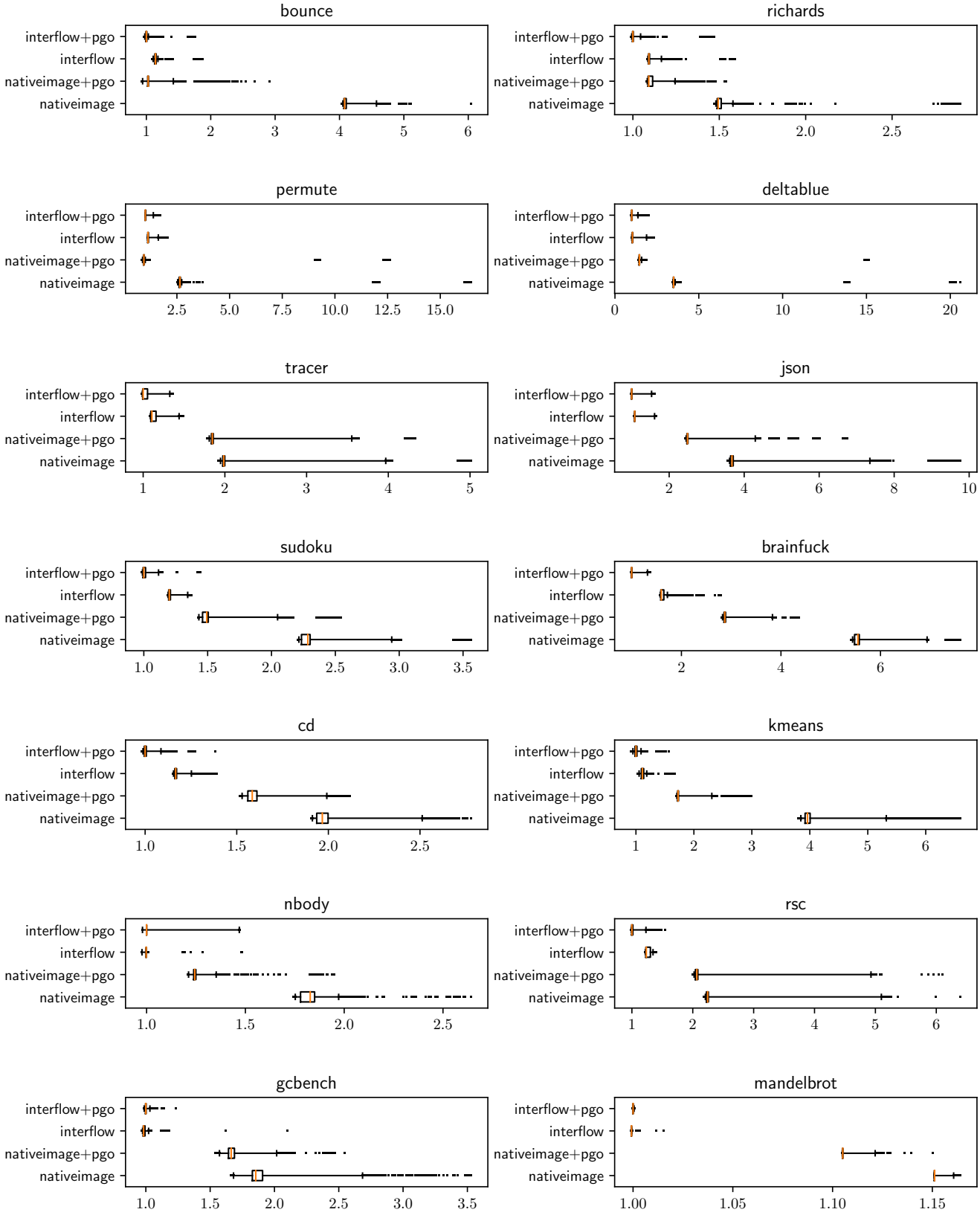


Figure 7.8 – Warmed-up performance relative to Native Image, less is better.

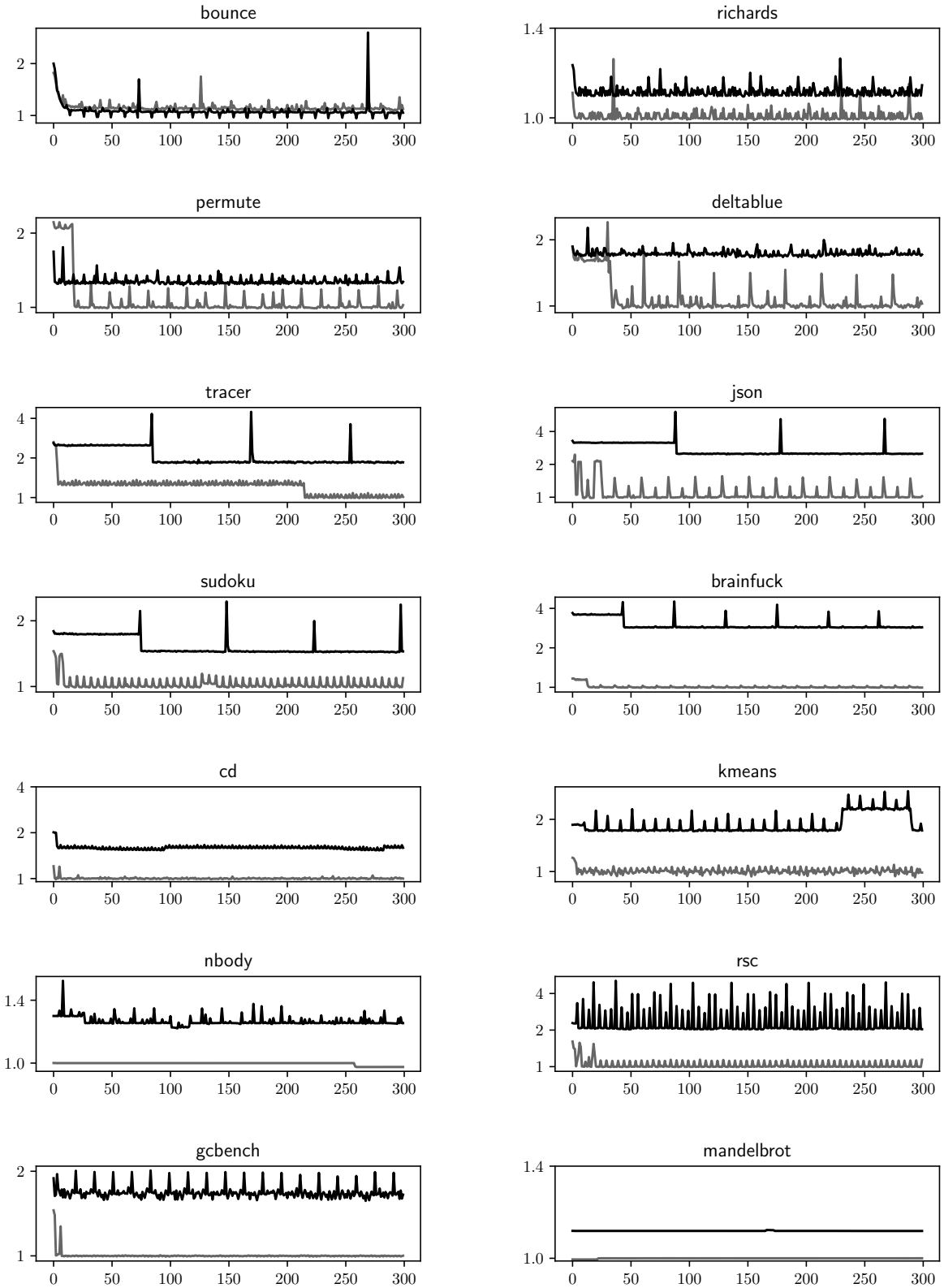


Figure 7.9 – Warm-up performance of Interflow with PGO (gray) relative to Native Image with PGO (black), less is better.

	bounce	richards	permute	deltablue	tracer	json	sudoku	brainfuck	cd	kmeans	nbody	rsc	gcbench	mandelbrot	geomean
interflow+pgo	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
interflow	1.14	1.09	1.13	1.04	1.11	1.08	1.20	1.60	1.17	1.11	1.00	1.24	0.98	1.00	1.13
jvm-pargc	0.87	0.91	2.17	1.51	1.67	2.05	1.00	1.24	1.04	1.57	1.09	1.22	1.12	1.07	1.27
jvm-g1gc	0.88	1.29	3.28	1.97	1.77	2.14	1.04	1.33	1.10	1.72	1.09	1.37	1.11	1.07	1.42

Figure 7.10 – Warm HotSpot JVM running time at 50 percentile, normalized by Interflow with PGO, less is better.

7.8 Performance relative to HotSpot JDK

As our final comparison, we compare Interfow's performance with HotSpot JVM after warm-up (Figures 7.10, 7.11).

HotSpot JVM offers several built-in garbage collectors. Parallel GC is the standard high-throughput garbage collector that aims to provide the best performance at the expense of the collection pause times. G1 GC [34] is a concurrent collector aims to offer decreased garbage collector pauses at the expense of the peak throughput.

In comparison to the HotSpot with Parallel GC, the purely static code produced by Interflow in combination with Commix GC offers 12% better geomean running time across our benchmark suite. Using purely static optimization techniques, we are not able to match the HotSpot performance on bounce, richards, brainfuck, cd, sudoku, and rsc benchmarks. Nevertheless, kmeans and permute show significant performance improvements even with purely static optimizations.

With the addition of profile-guided optimization, the geomean indicates 27% better performance. We are able to surpass HotSpot on all benchmarks except bounce and richards.

An advantage of ahead-of-time compiled code comes from quick warm-up compared to the just-in-time compiled runtime such as HotSpot JVM (Figure 7.12). The initial iterations can have over 8x faster runtime performance before the code is fully optimized by the runtime. AOT compilation offers a significant performance increase for short-lived applications such as rsc where the JIT warm-up overhead affects the performance of a cold run.

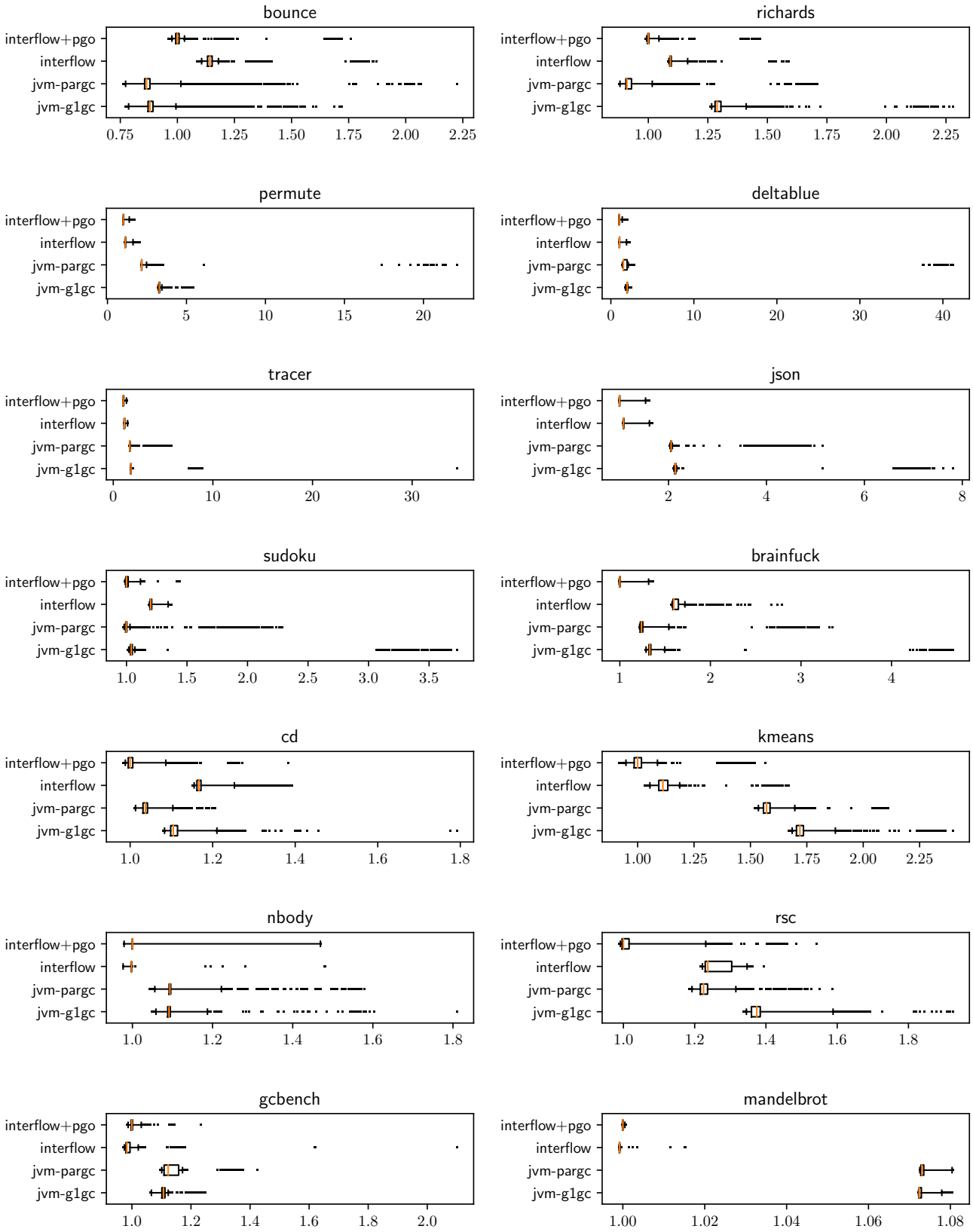


Figure 7.11 – Warmed-up performance relative to HotSpot JDK, less is better.

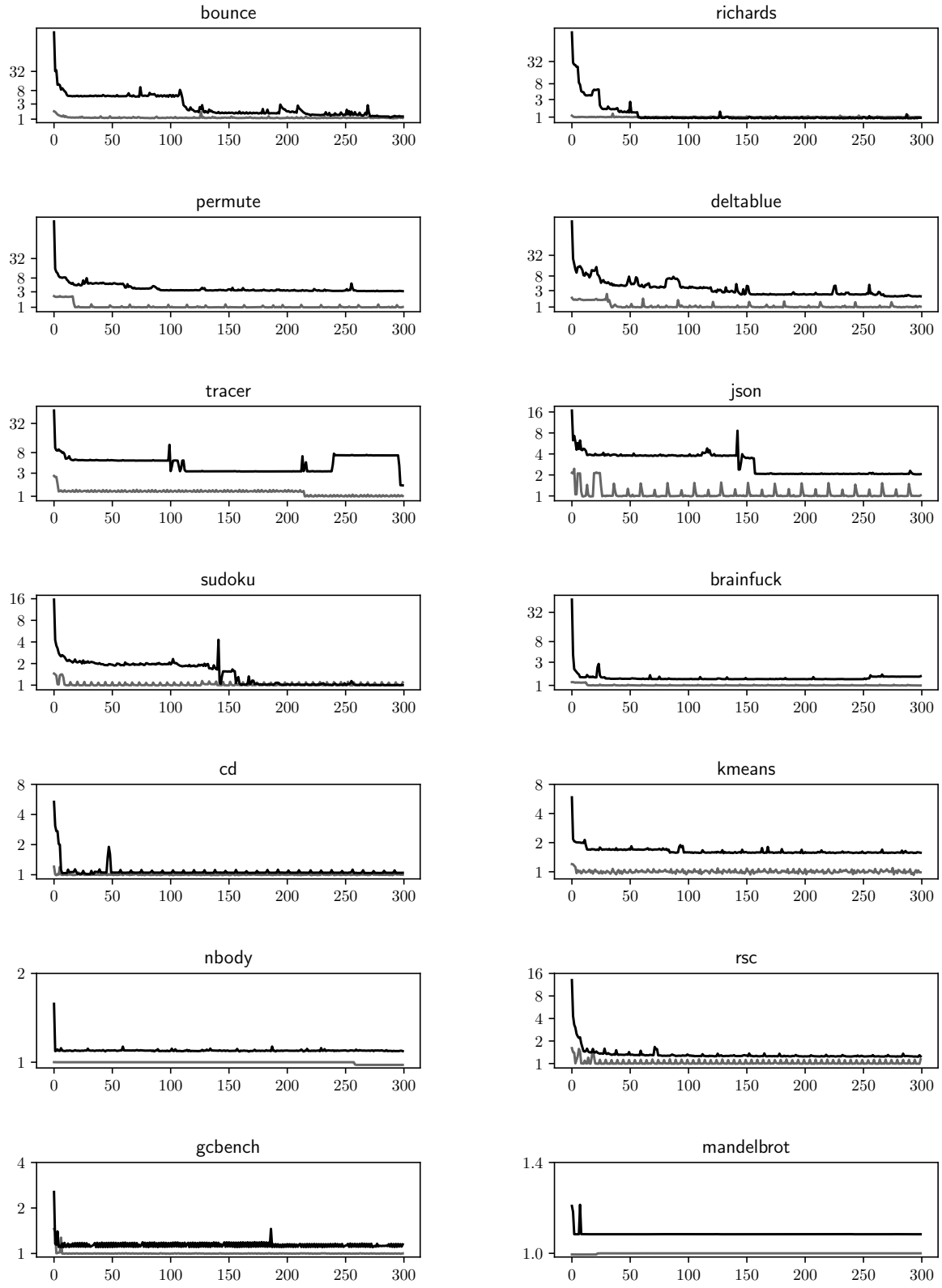


Figure 7.12 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black), less is better.

7.9 Conclusion

In this section, we evaluated the performance of the compiled code emitted by the Scala Native toolchain.

We have evaluated the impact of our optimizations against the baseline compiled code. Interflow offers a significant performance uplift based on its purely static optimization pipeline. The cost of the performance comes at the risk of increased binary size as seen on Rsc benchmark.

We observed that while the runtime impact of PGO is less pronounced than the impact of Interflow, it is still necessary to achieve the best throughput on some of the benchmarks. The most significant impact of profile feedback lies in better binary size thanks to improved inlining and method duplication decisions. In fact, with profile information, we can produce code that consistently beats baseline compilation on size and offers only a slightly binary size increase compared to the conservative version of our optimizer.

Moreover, our evaluation shows that results demonstrated by Interflow in combination with PGO outperform the HotSpot JVM as well as Native Image on most of our benchmarks.

8 Conclusion

In this thesis, we presented the complete design and implementation of the Scala Native optimizing compiler that fulfills the following design goals (Section 1.1):

1. Startup time.
2. Peak performance.
3. Compatibility.

To achieve these goals, we rely on a whole-program optimizing compilation model.

First, we introduced a design for flow-sensitive optimizer called Interflow. It relies on a single graph-free traversal of the whole program to perform a number of optimizations including method duplication, partial evaluation, allocation sinking and inlining in a fused single-pass traversal of the whole program.

Moreover, we extended the Interflow with support for profile feedback obtained through runtime instrumentation. We proposed white-gray code splitting as an underlying framework to reason about JIT-style speculative optimizations in the ahead-of-time setting. These techniques allow us to selectively optimize hot paths of the programs while ignoring the cold paths and optimizing them for size rather than the best runtime performance.

Finally, we evaluated the runtime performance of our implementation and compared it against Graal Native Image and HotSpot JVM. Our evaluation suggests that our implementation outperforms existing AOT compilers for Java. Furthermore, with the addition of the profile-guided optimizations, we are able to outperform HotSpot JVM on most of our benchmarks.

A Raw Benchmark Results

In this appendix, we provide charts for complete runs of Interflow with PGO as compared to HotSpot JDK and Native Image. The data here is presented without any post processing and corresponds to direct measurements as obtained during our performance evaluation.

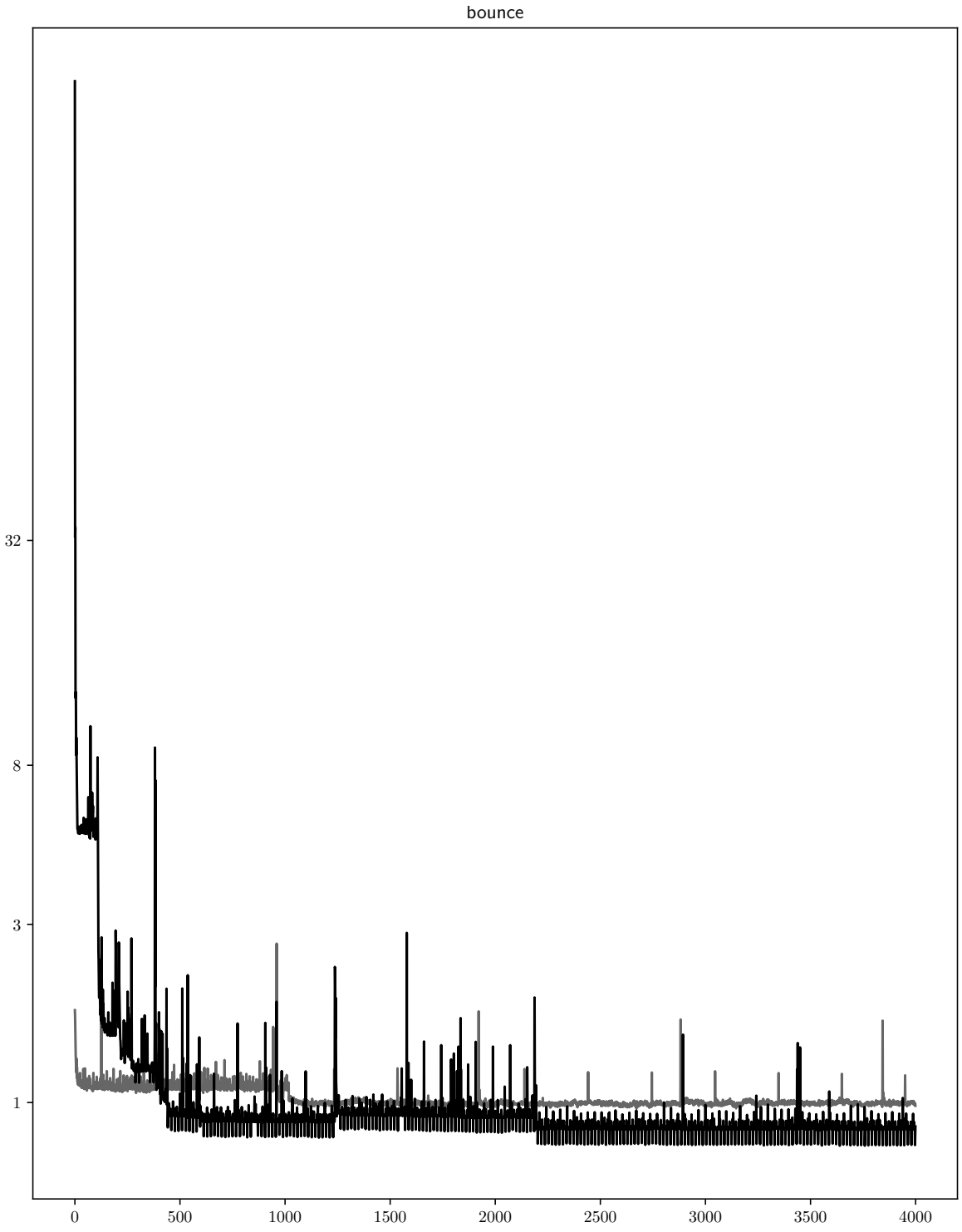


Figure A.1 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the bounce benchmark, less is better.

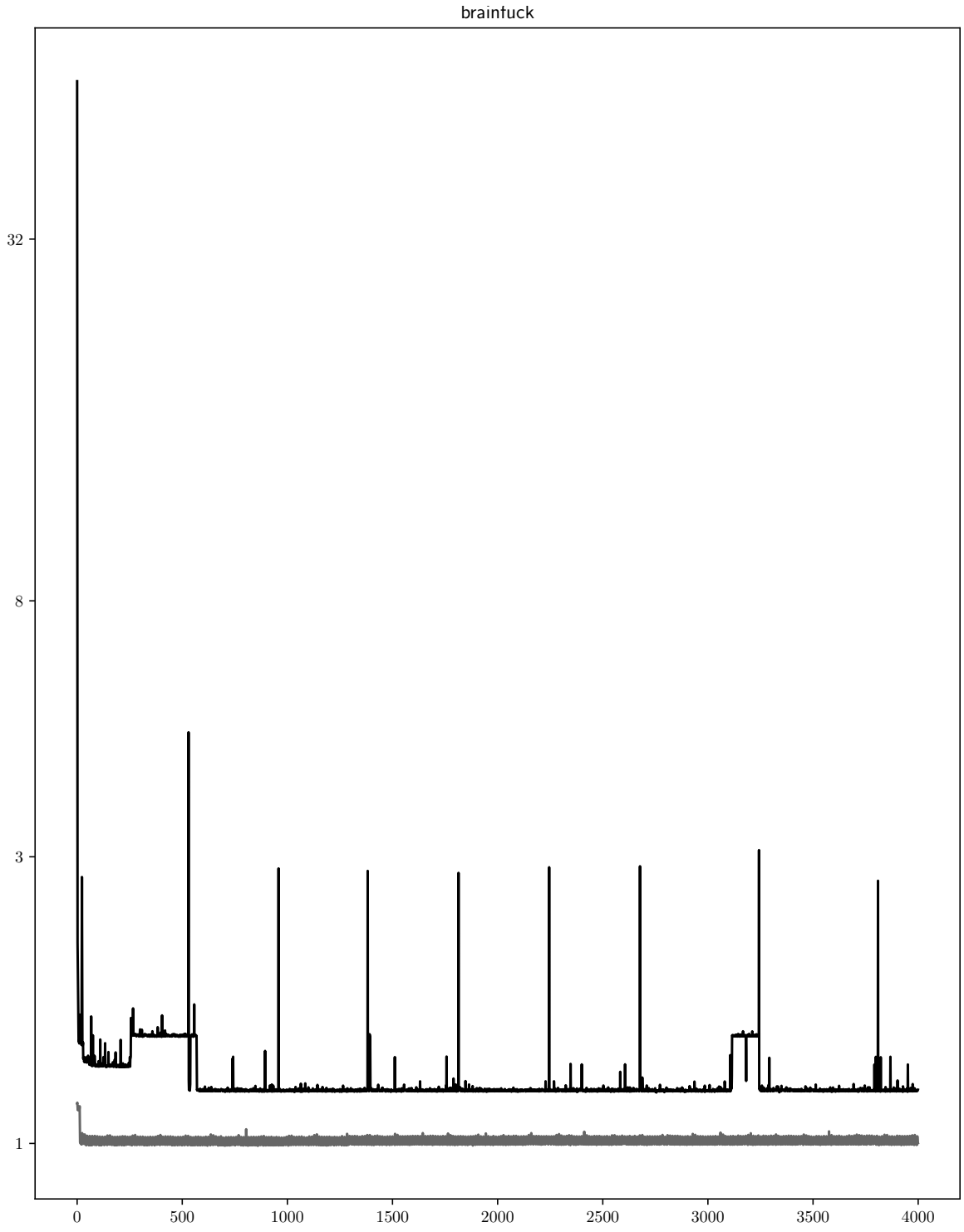


Figure A.2 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the brainfuck benchmark, less is better.

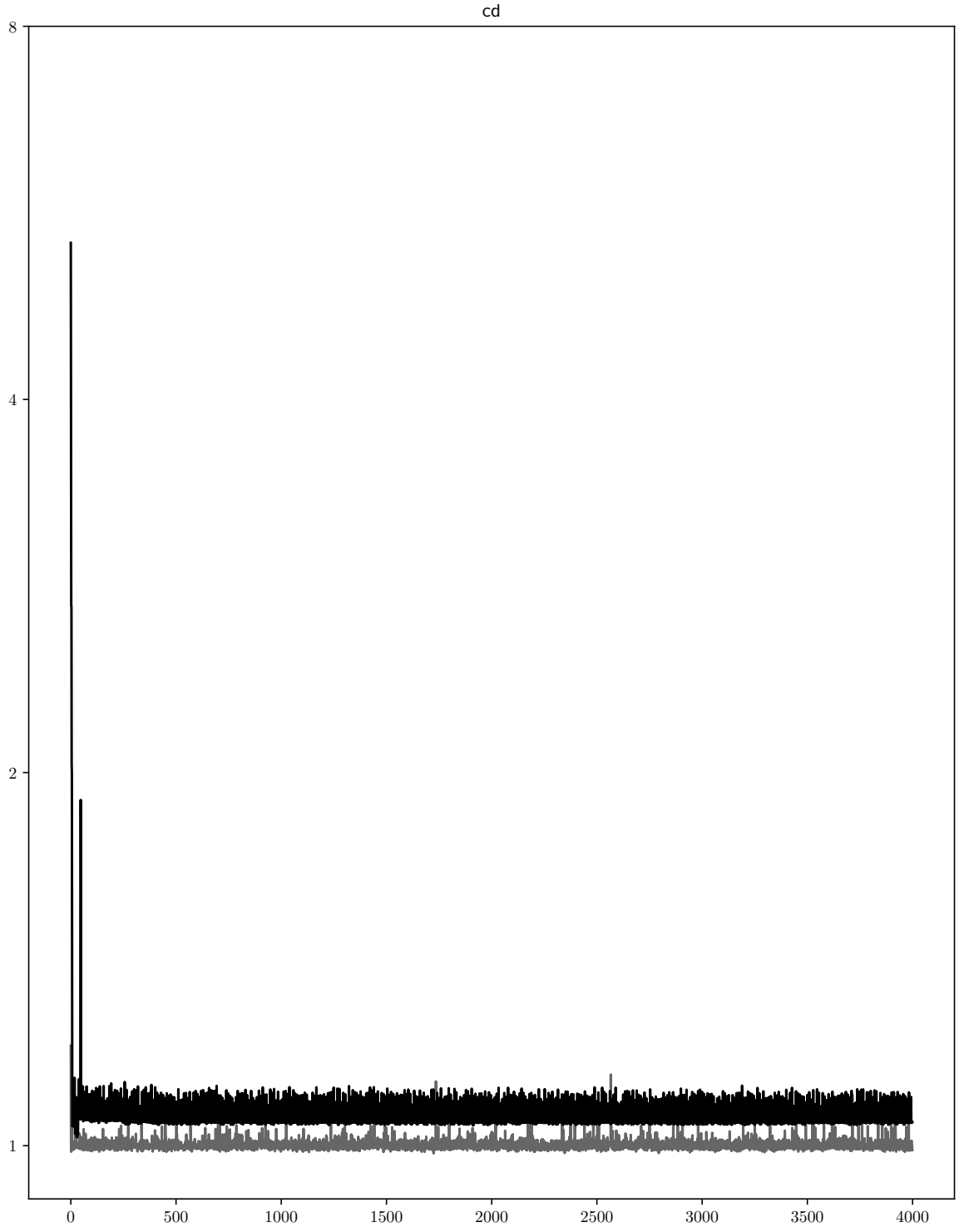


Figure A.3 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the cd benchmark, less is better.

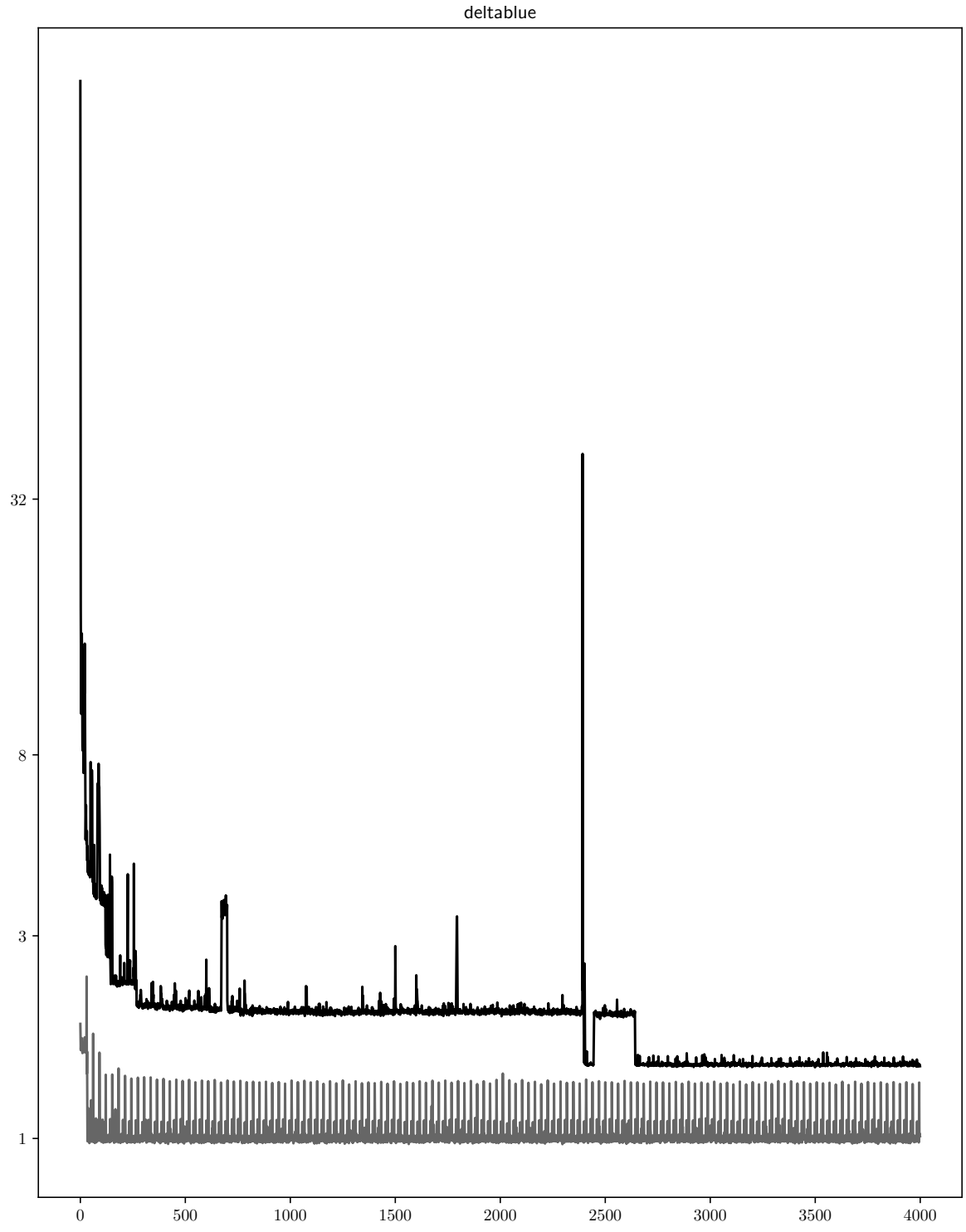


Figure A.4 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the deltablue benchmark, less is better.

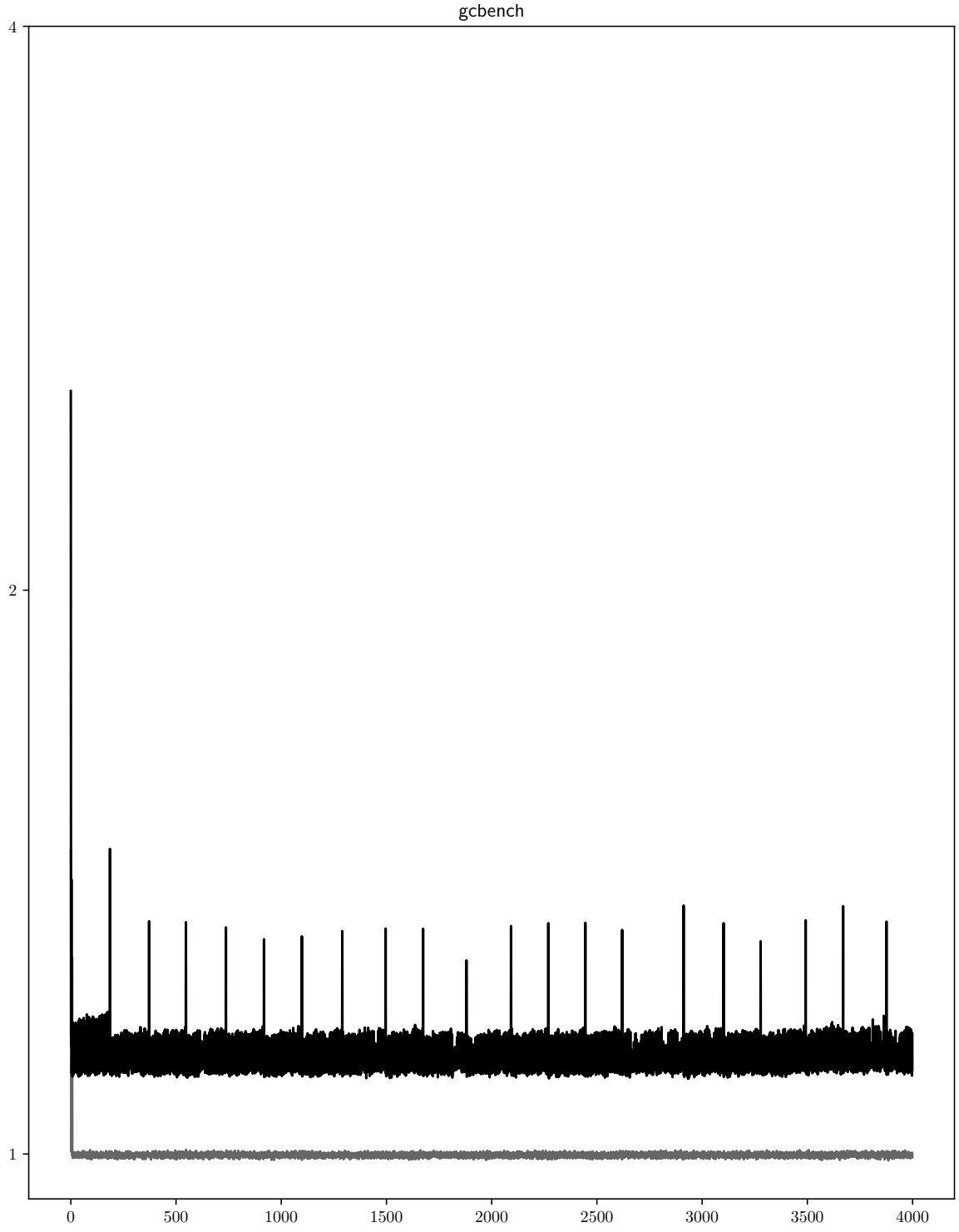


Figure A.5 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the gcbench benchmark, less is better.

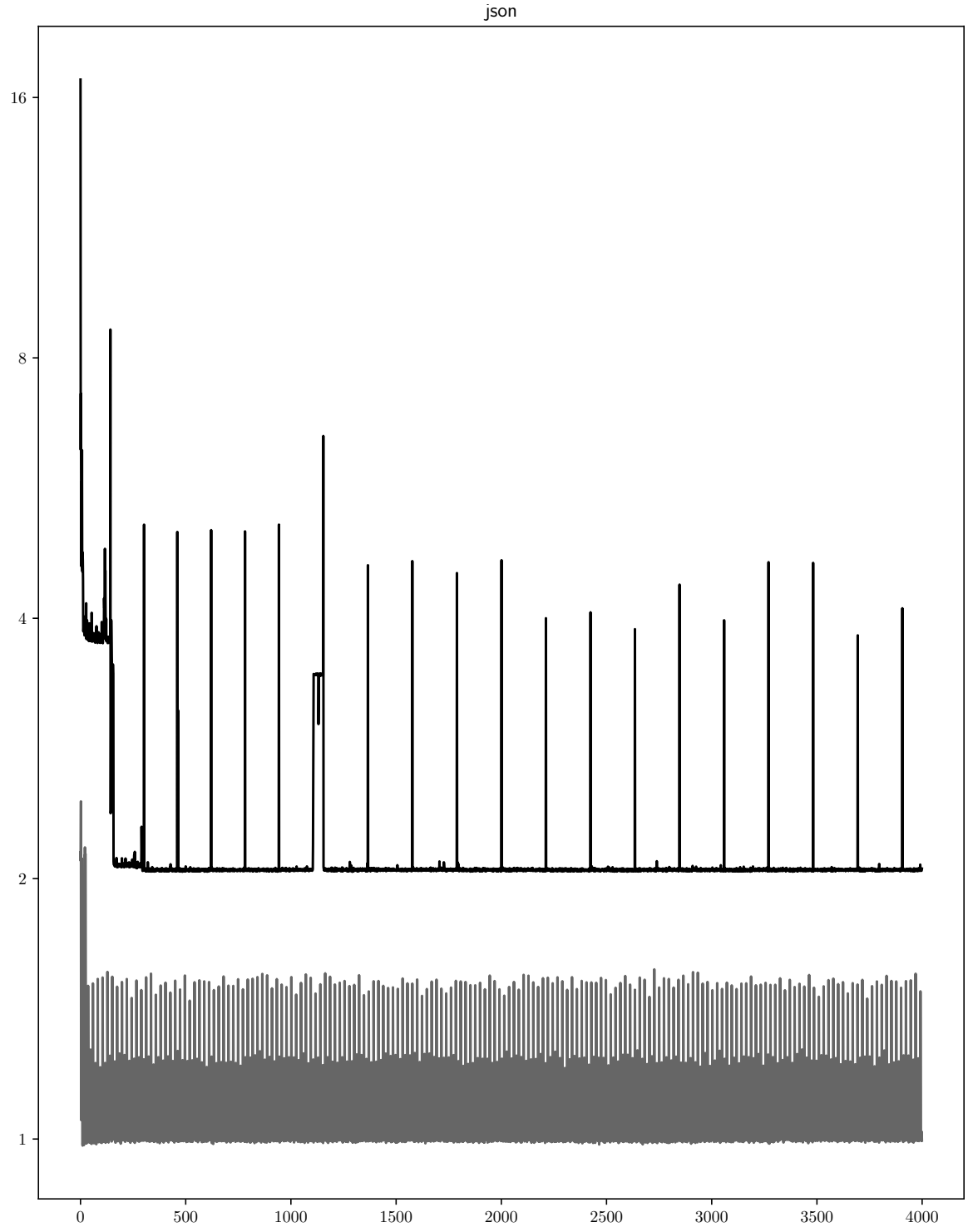


Figure A.6 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the json benchmark, less is better.

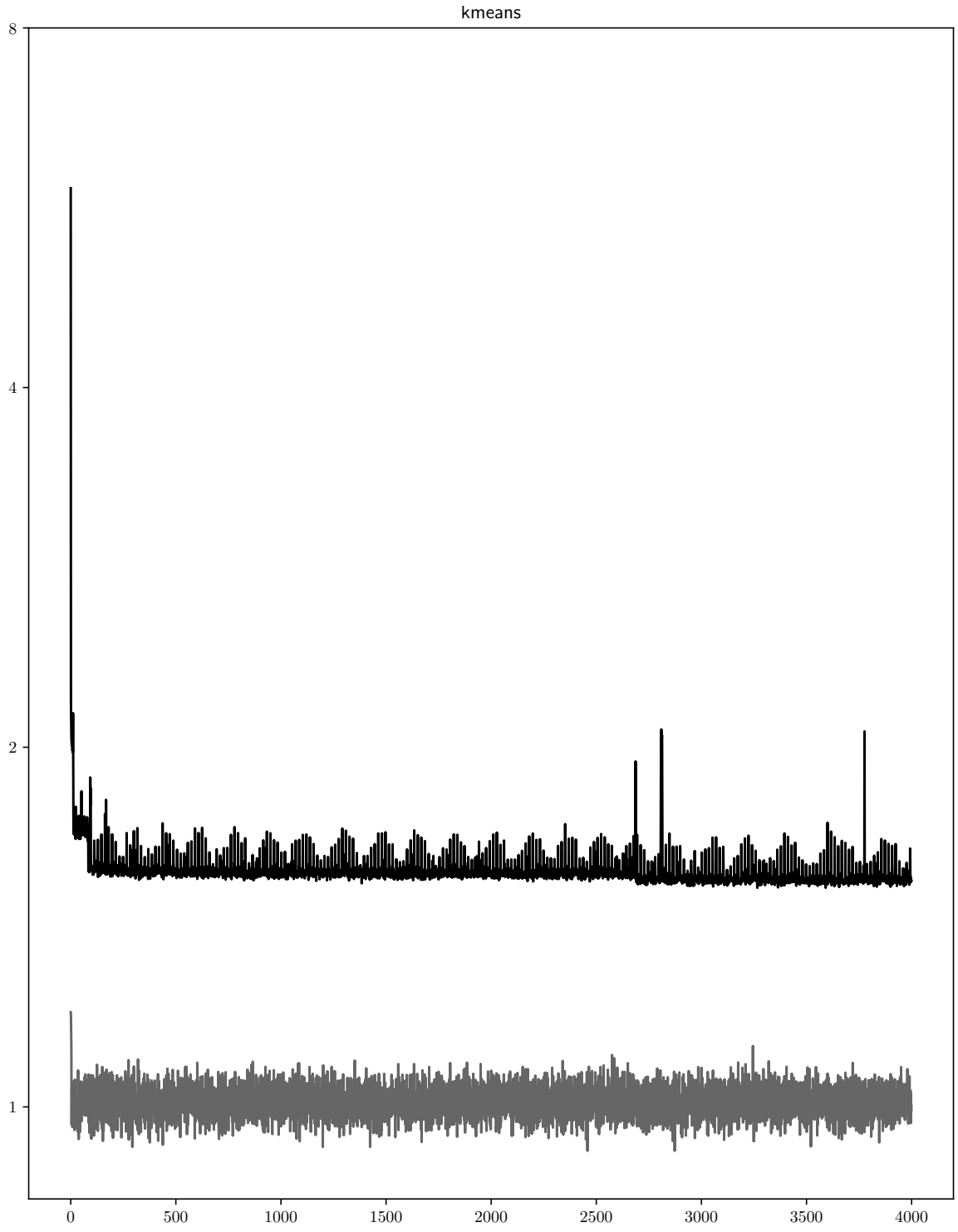


Figure A.7 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the kmeans benchmark, less is better.

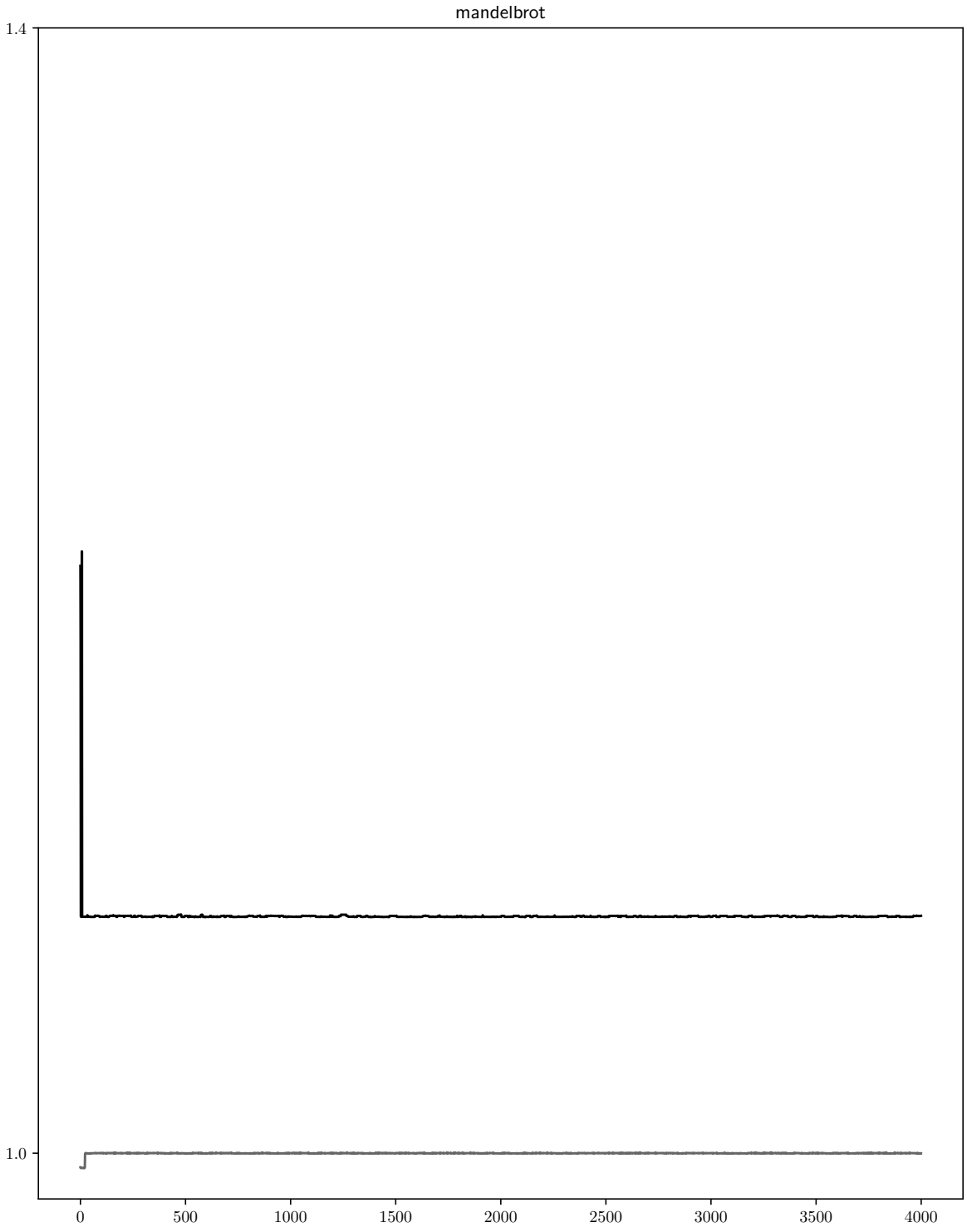


Figure A.8 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the mandelbrot benchmark, less is better.

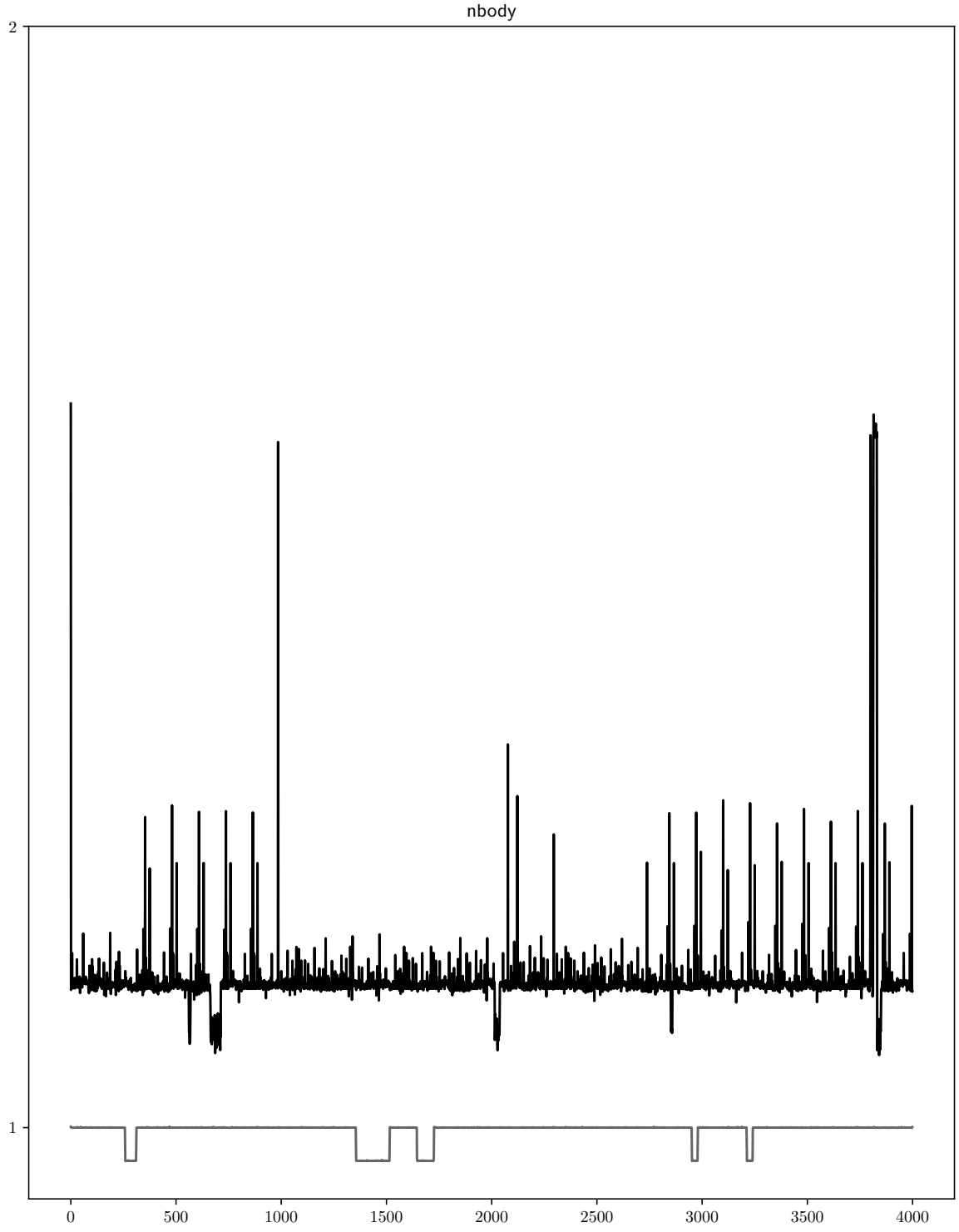


Figure A.9 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the nbody benchmark, less is better.

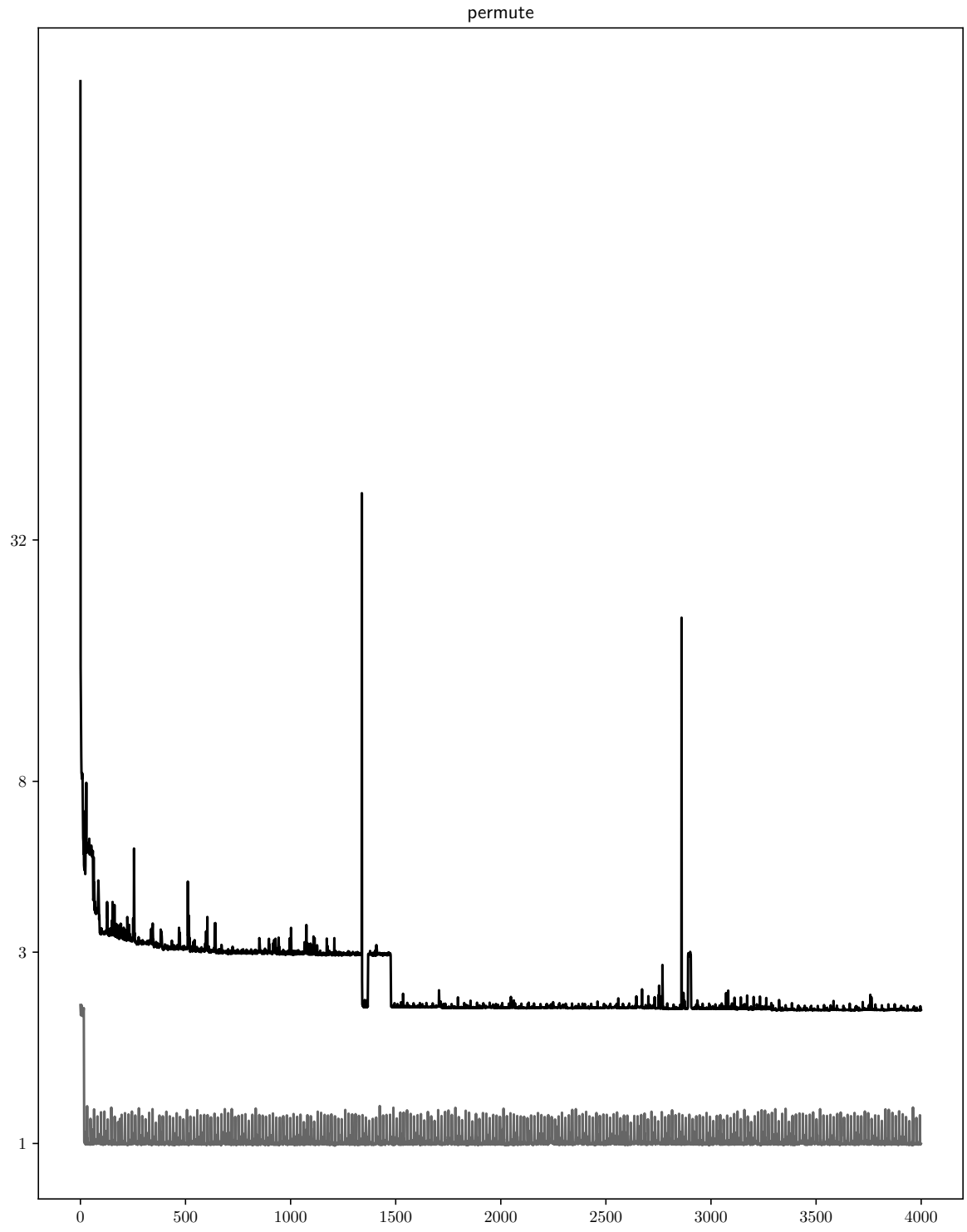


Figure A.10 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the permute benchmark, less is better.

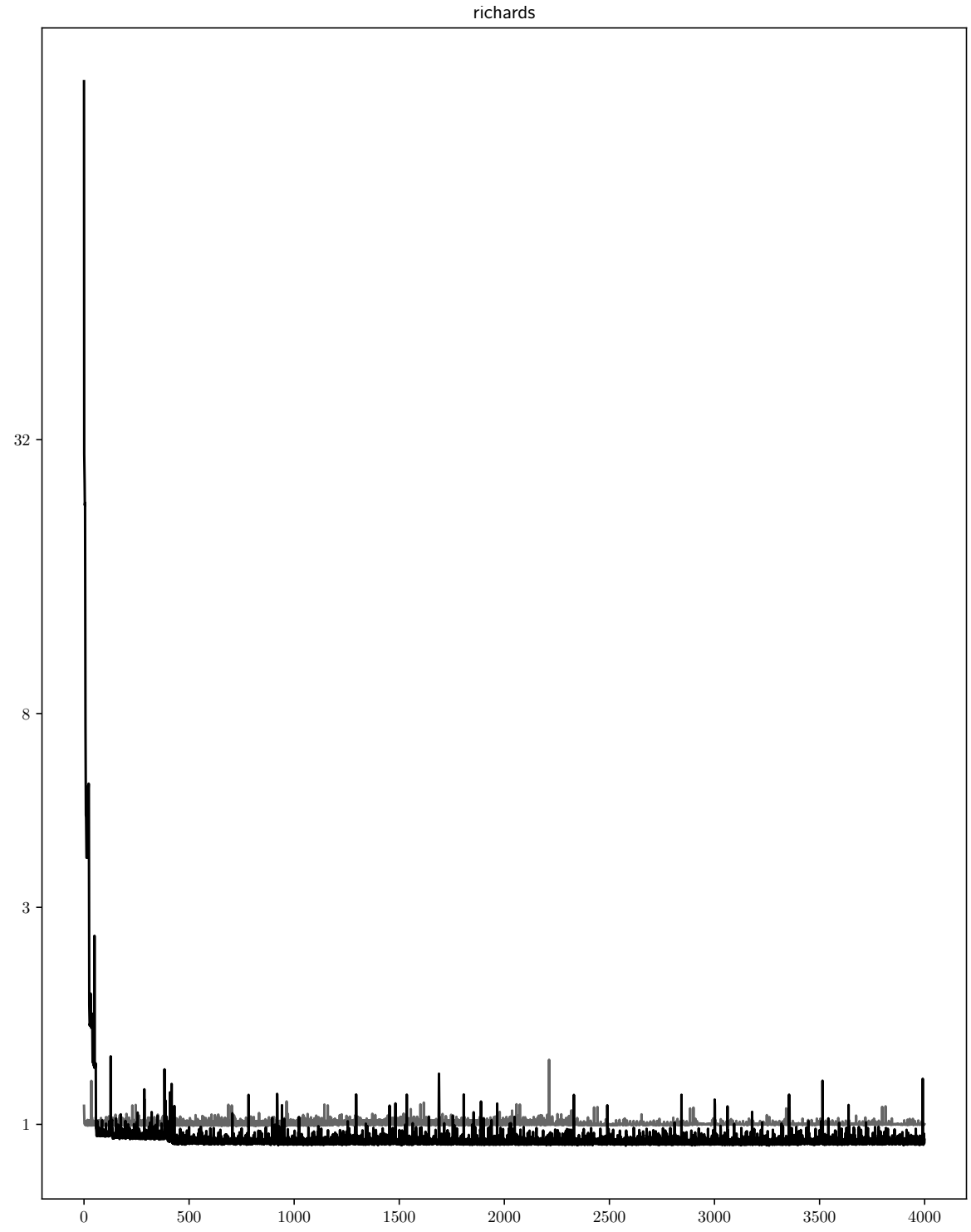


Figure A.11 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the richards benchmark, less is better.

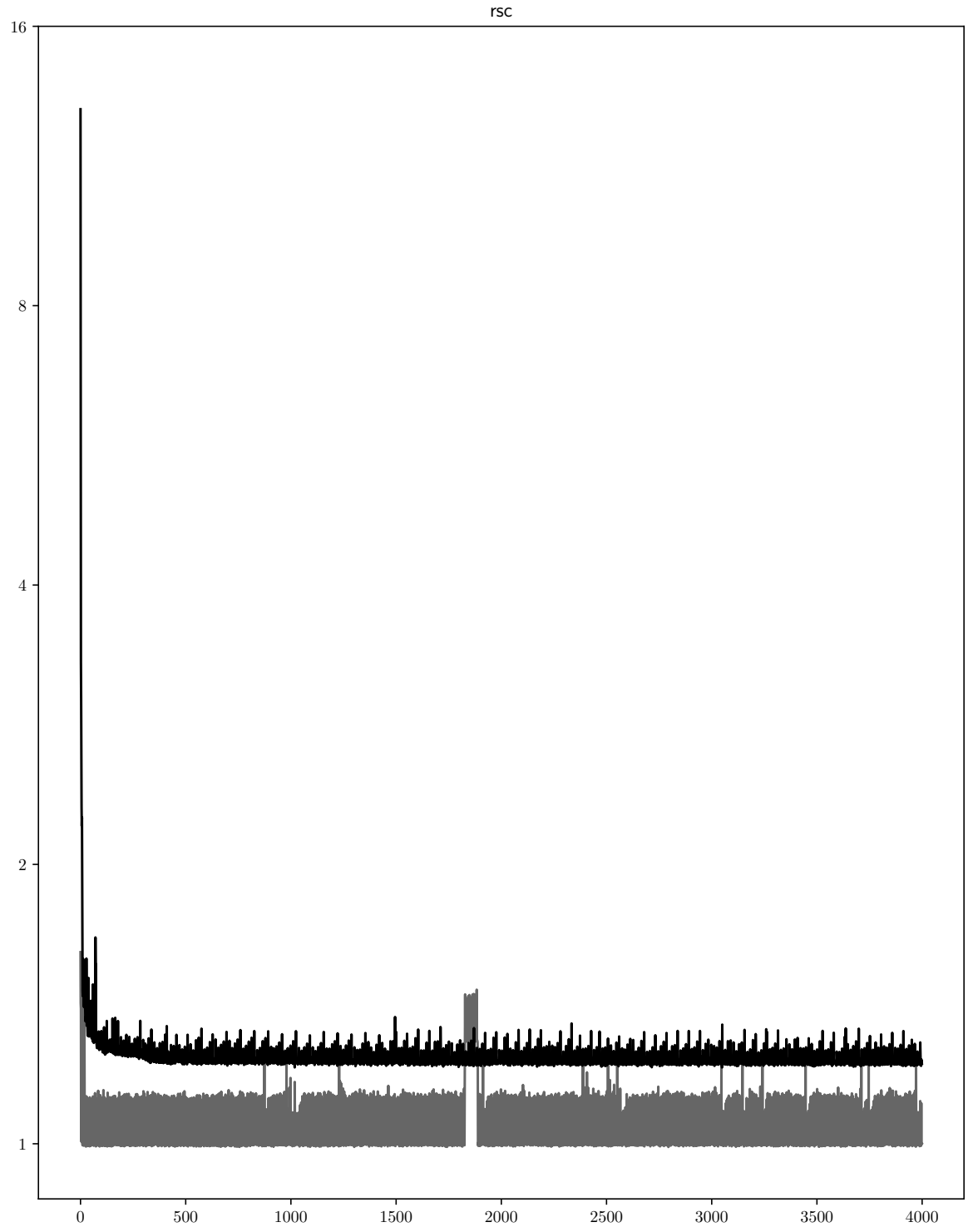


Figure A.12 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the rsc benchmark, less is better.

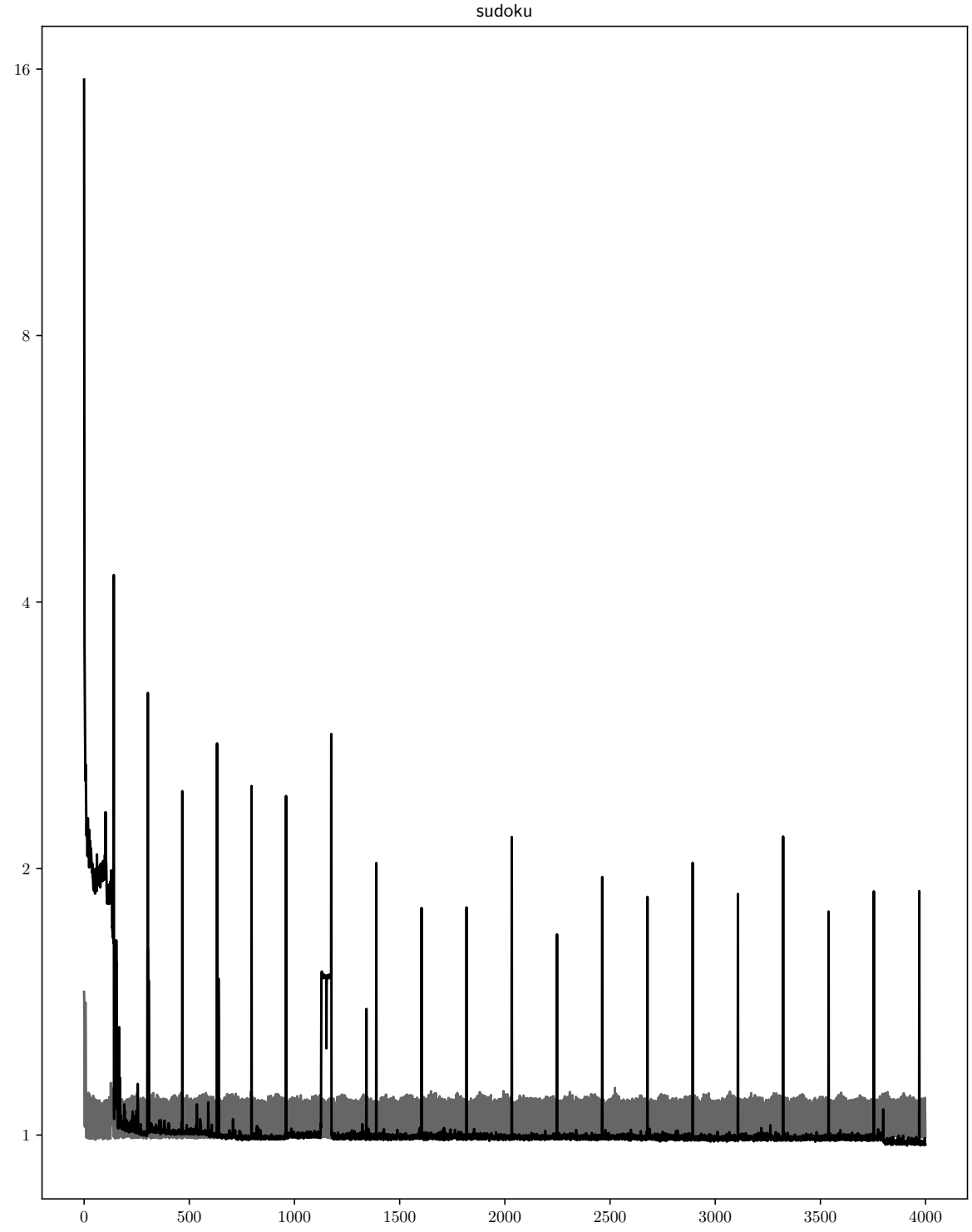


Figure A.13 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the sudoku benchmark, less is better.

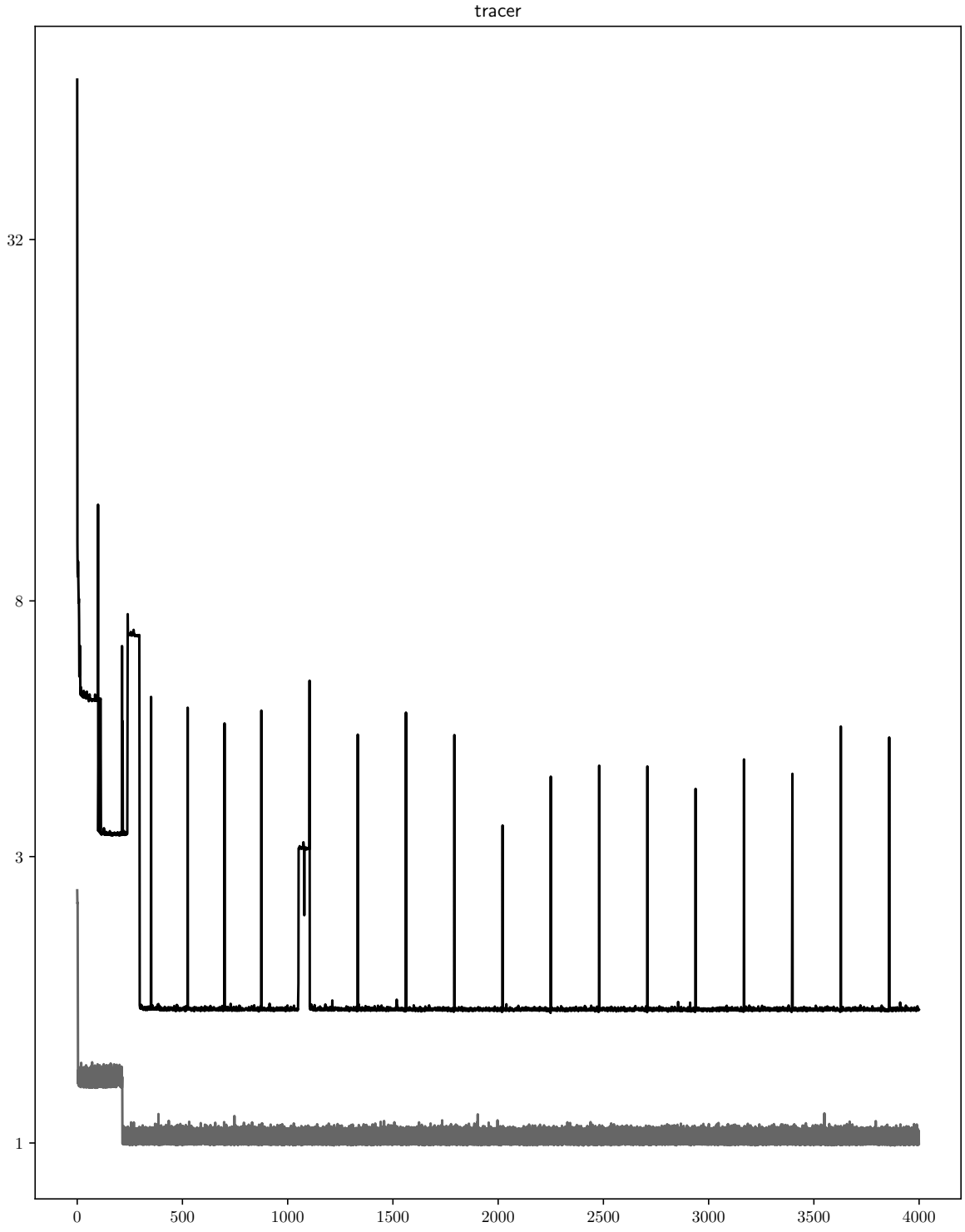


Figure A.14 – Warm-up performance of Interflow with PGO (gray) relative to HotSpot JDK with Paralell GC (black) on the tracer benchmark, less is better.

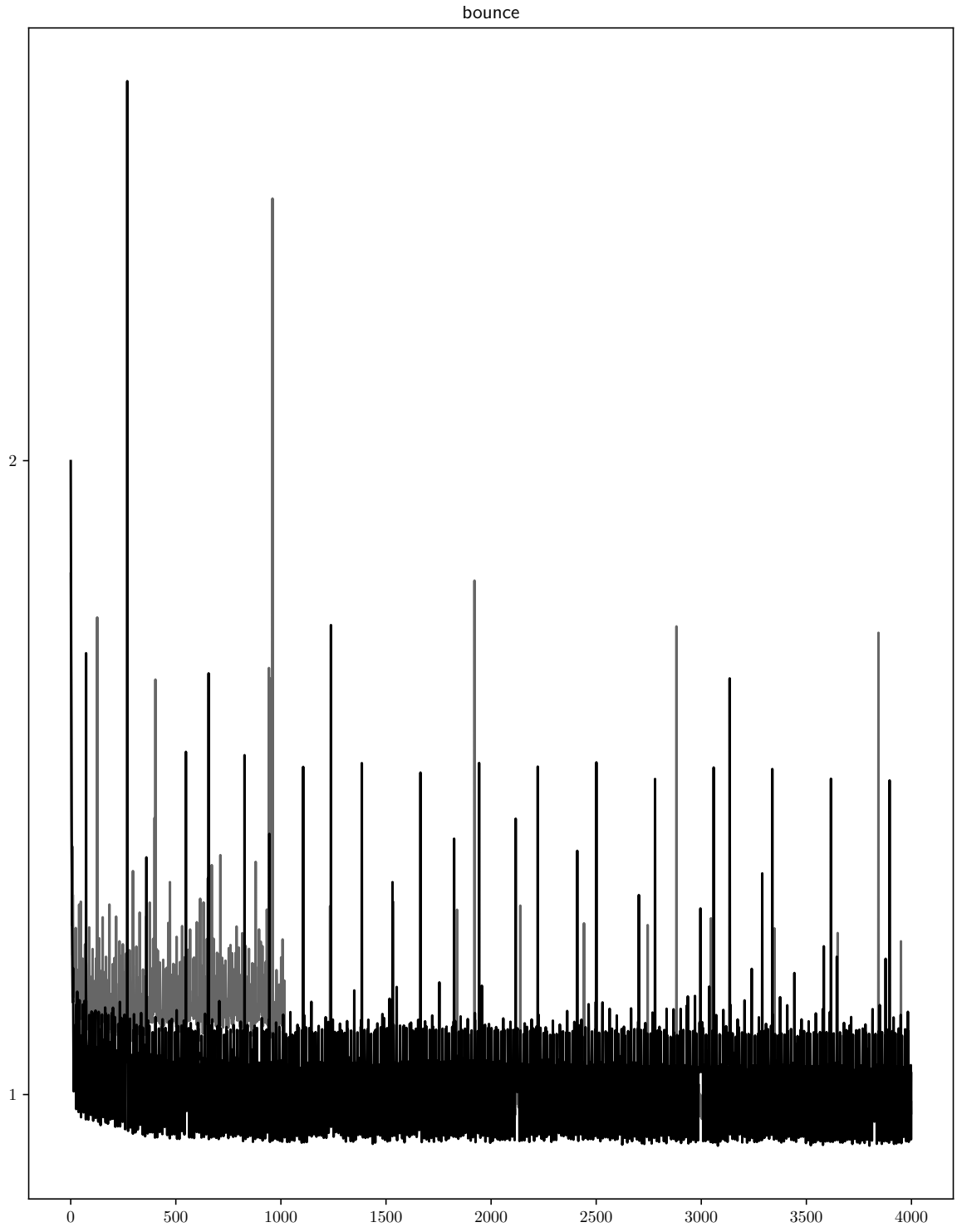


Figure A.15 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the bounce benchmark, less is better.

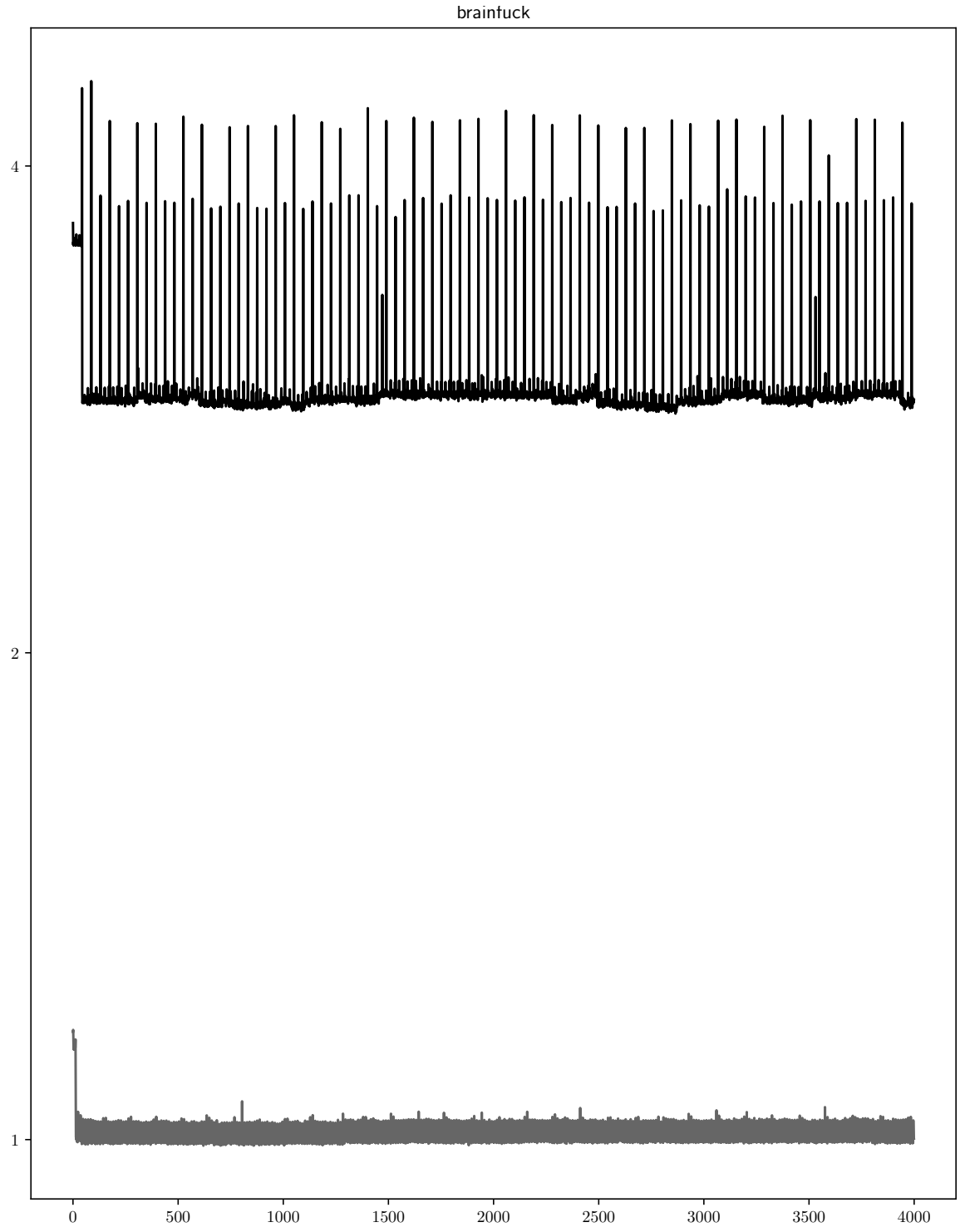


Figure A.16 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the brainfuck benchmark, less is better.

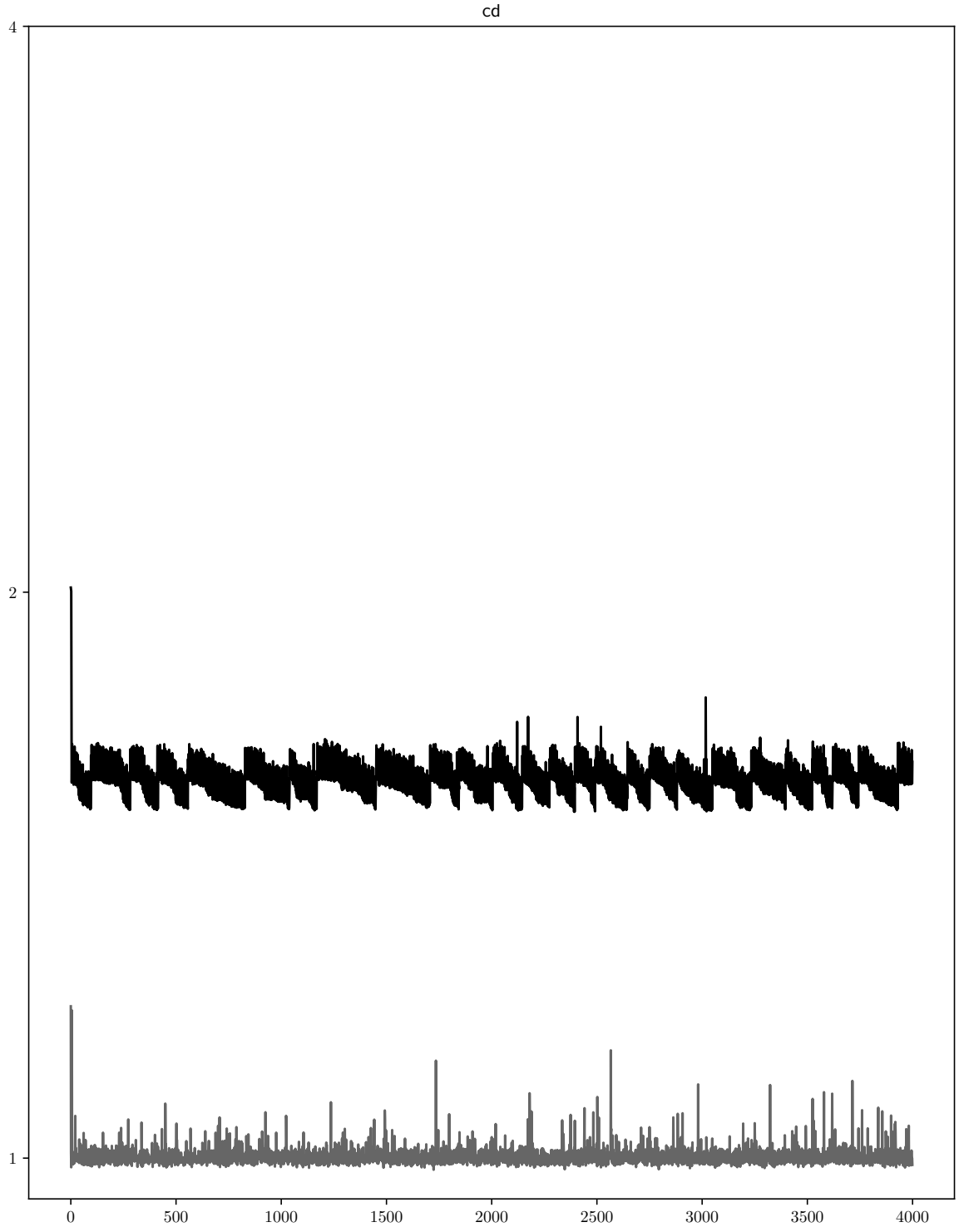


Figure A.17 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the cd benchmark, less is better.

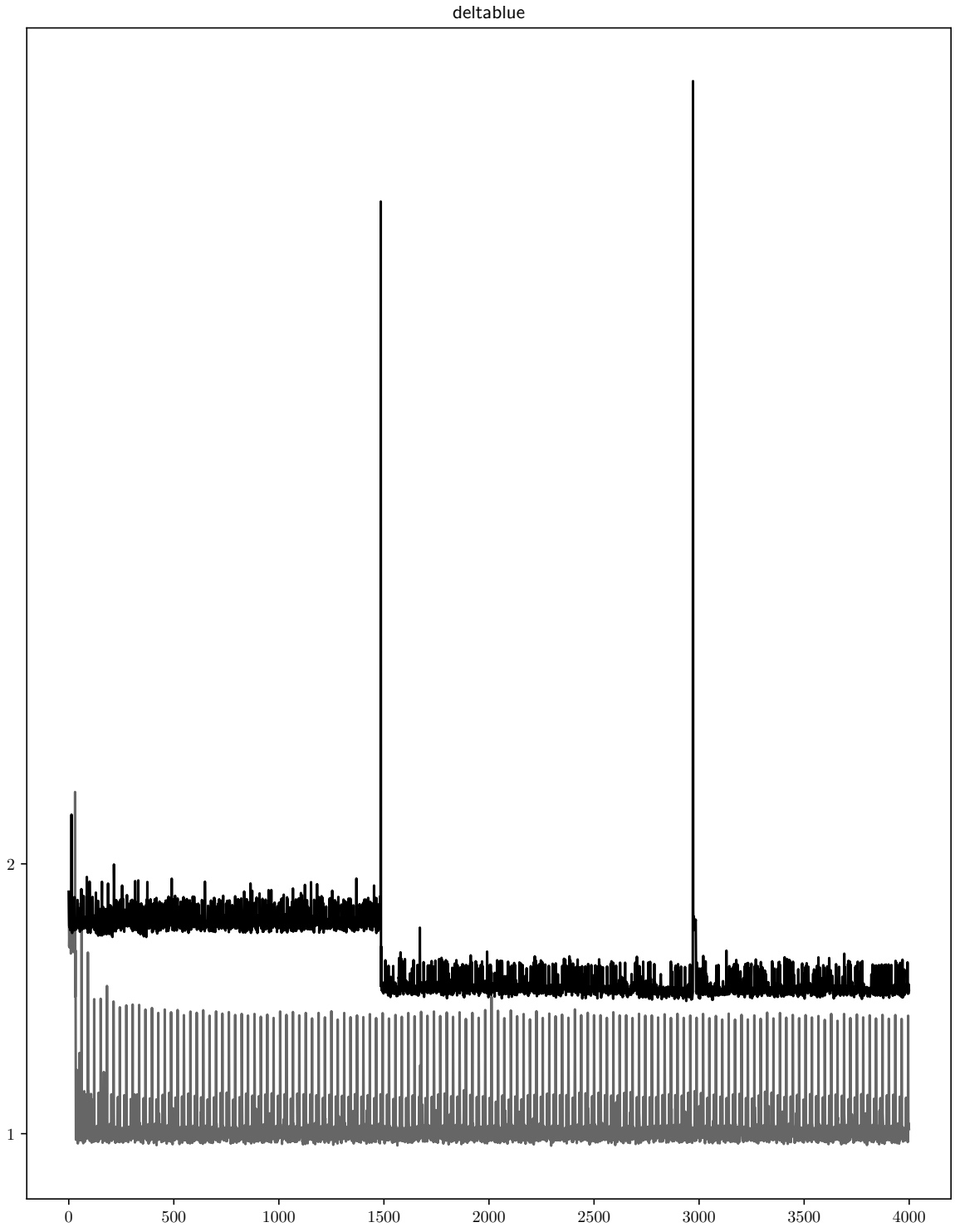


Figure A.18 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the deltablue benchmark, less is better.

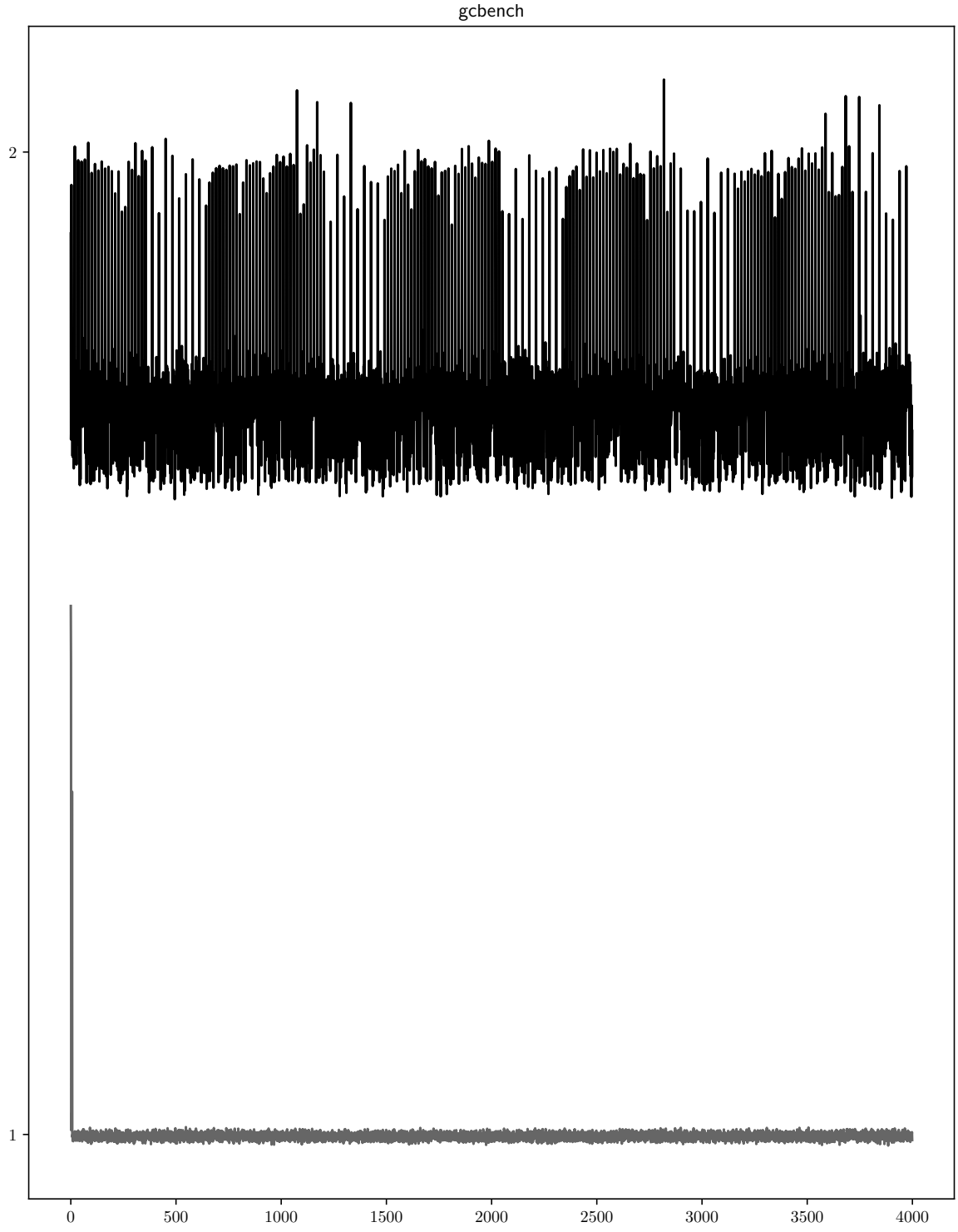


Figure A.19 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the gcbench benchmark, less is better.

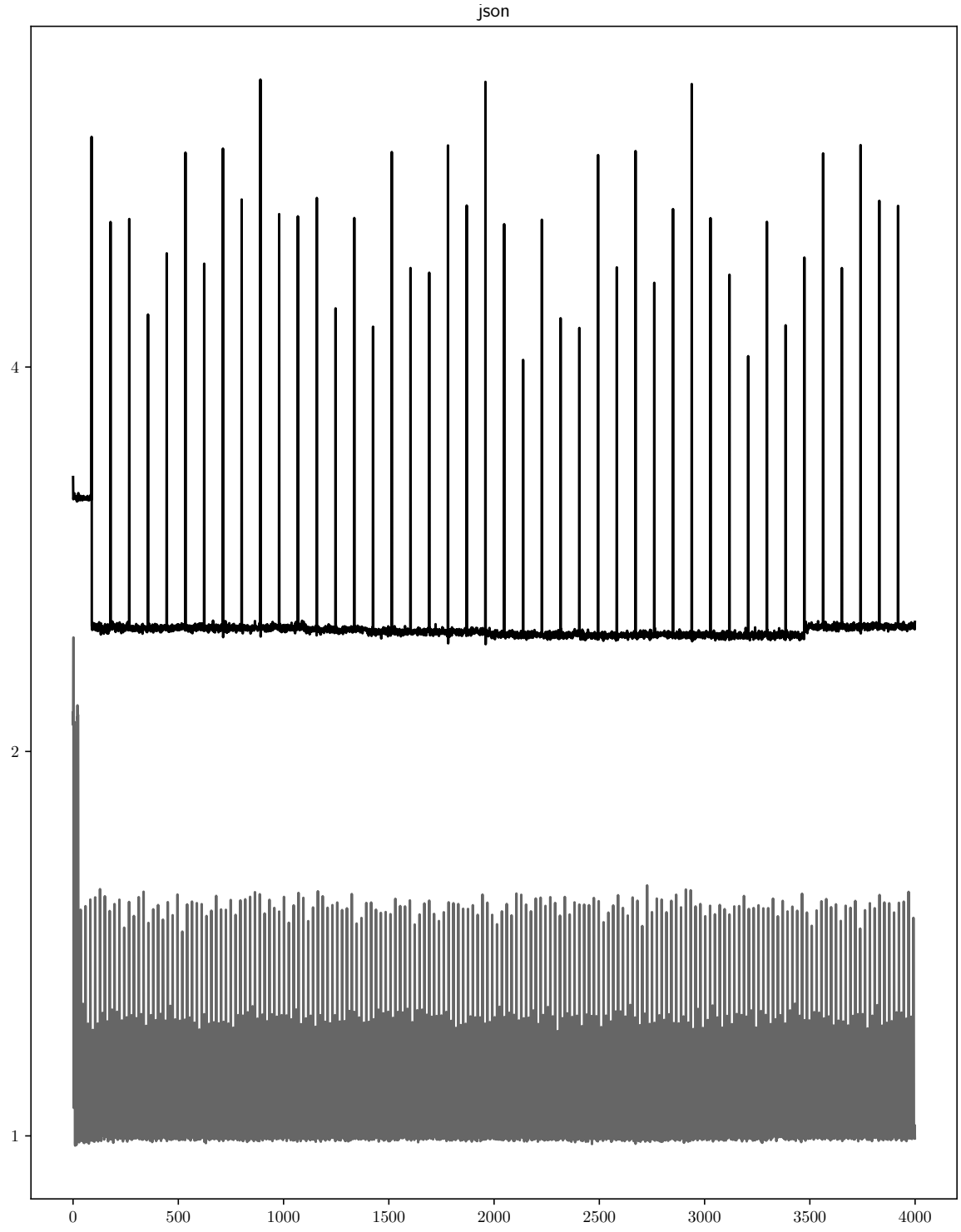


Figure A.20 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the json benchmark, less is better.

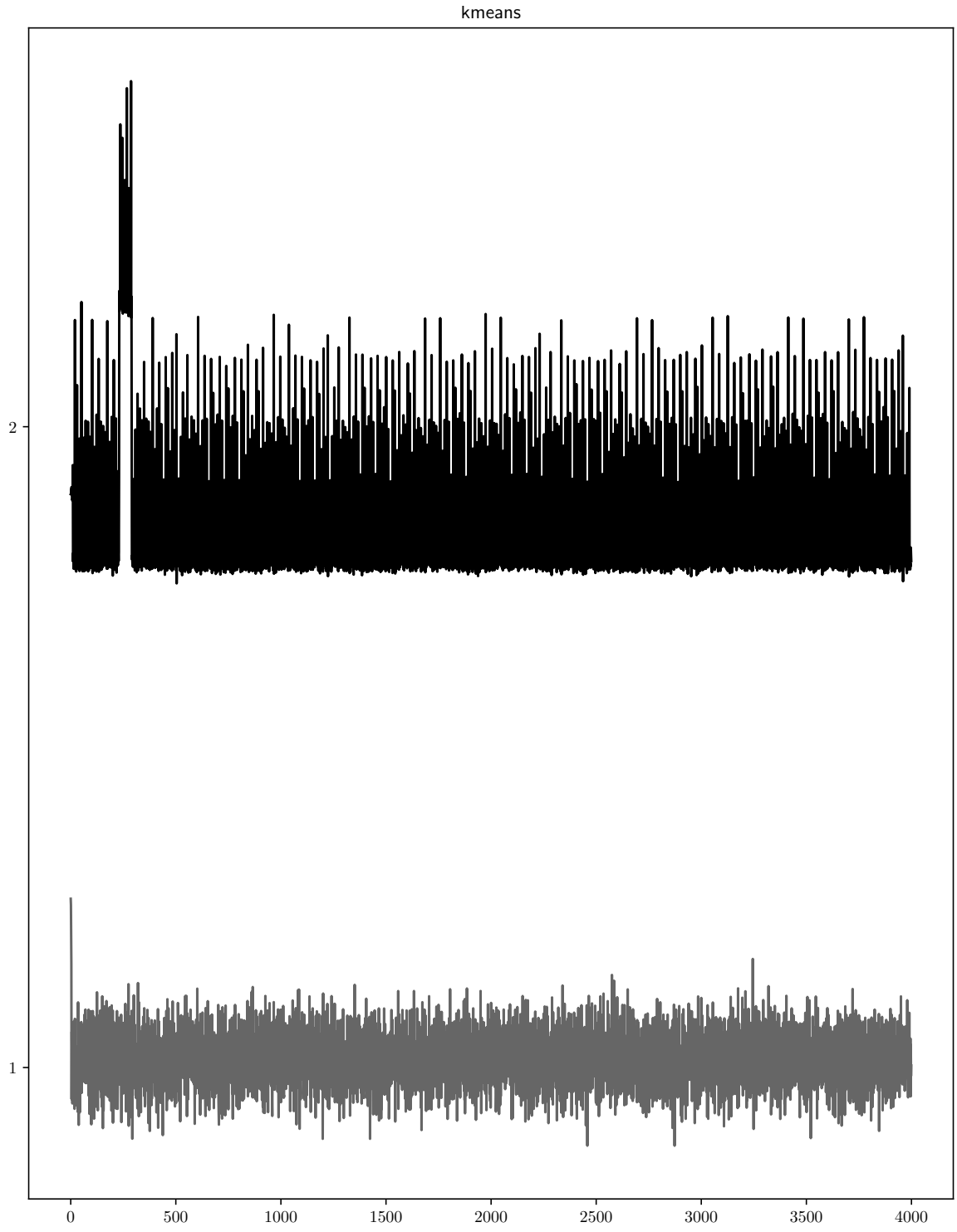


Figure A.21 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the kmeans benchmark, less is better.

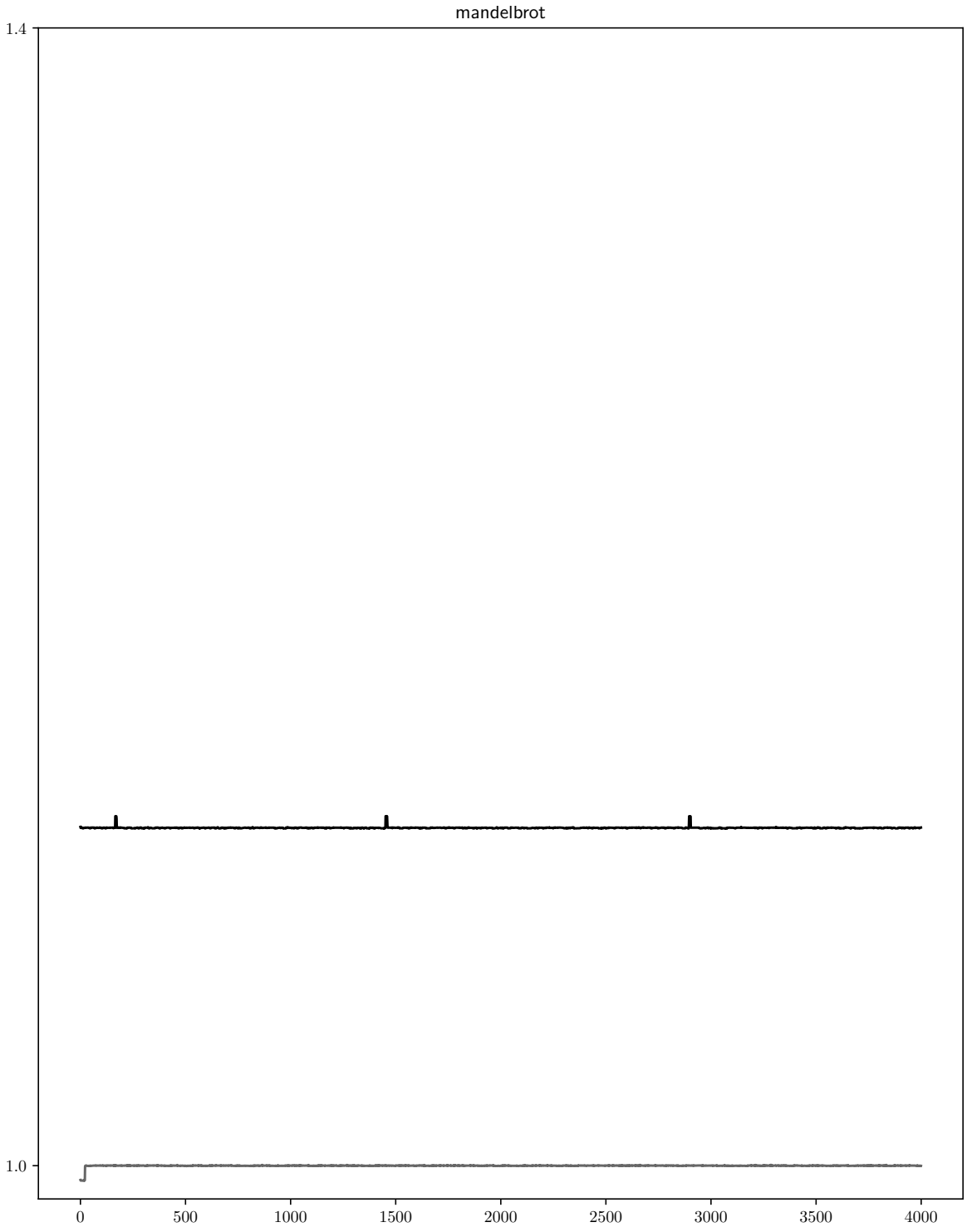


Figure A.22 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the mandelbrot benchmark, less is better.

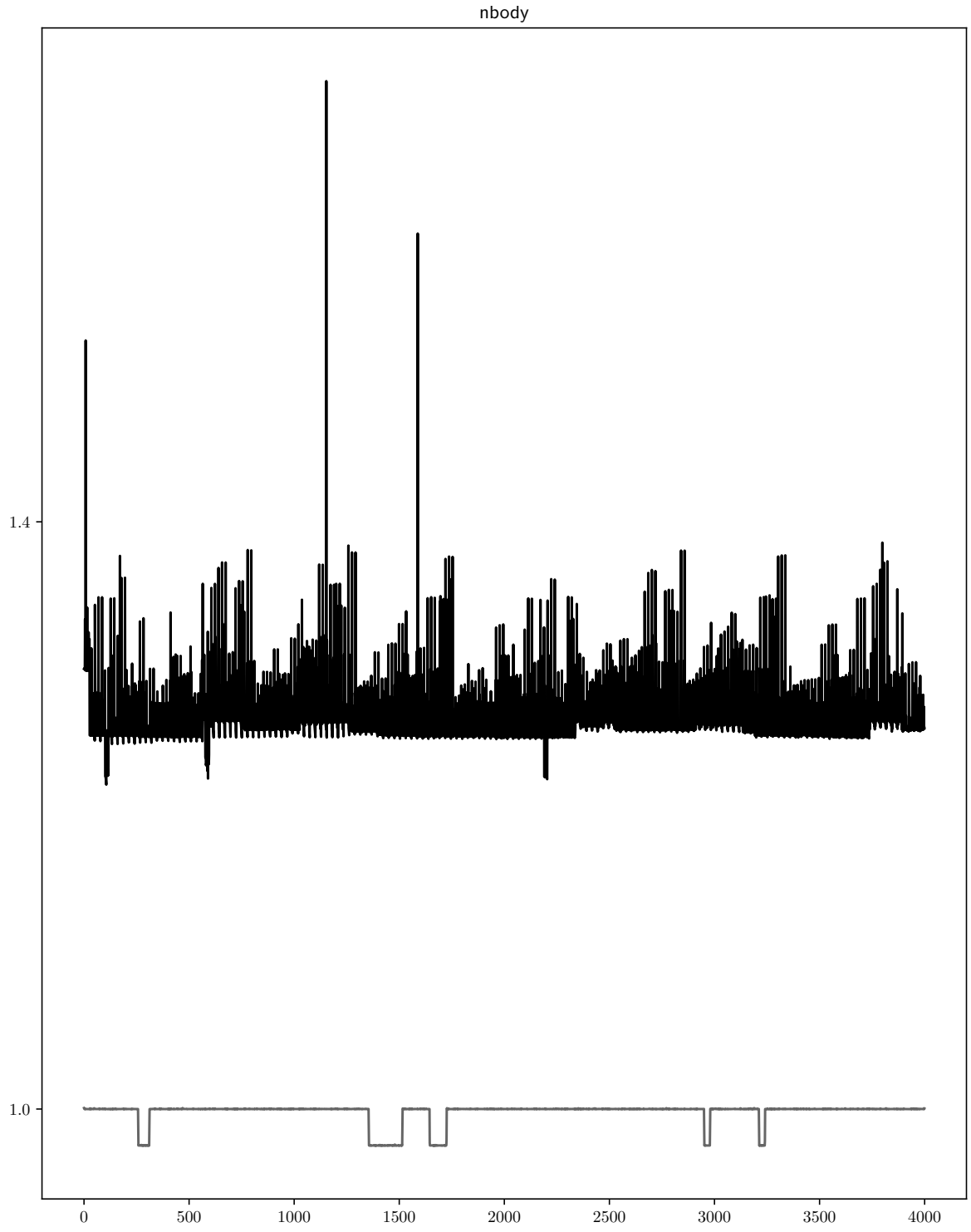


Figure A.23 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the nbody benchmark, less is better.

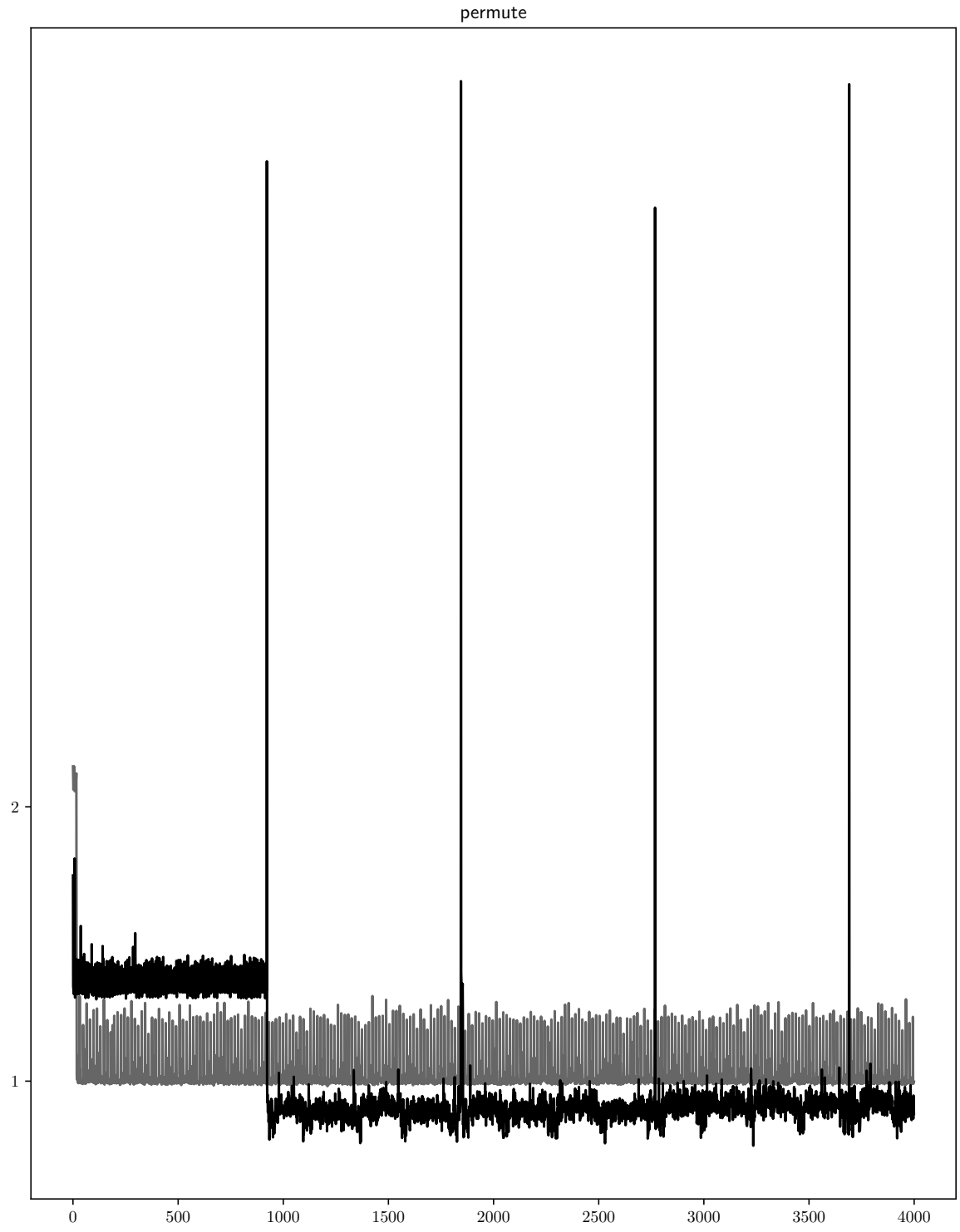


Figure A.24 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the permute benchmark, less is better.

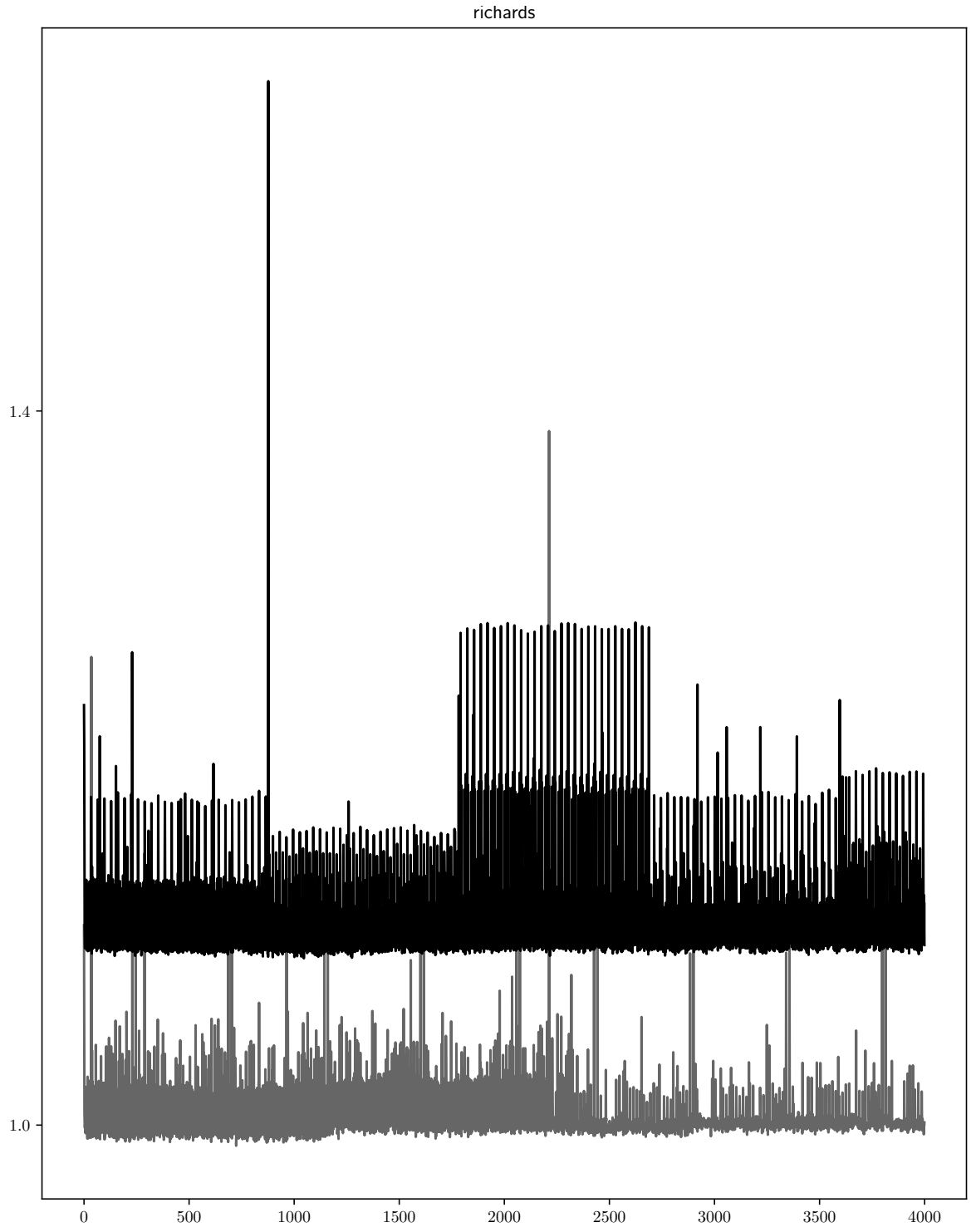


Figure A.25 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the richards benchmark, less is better.

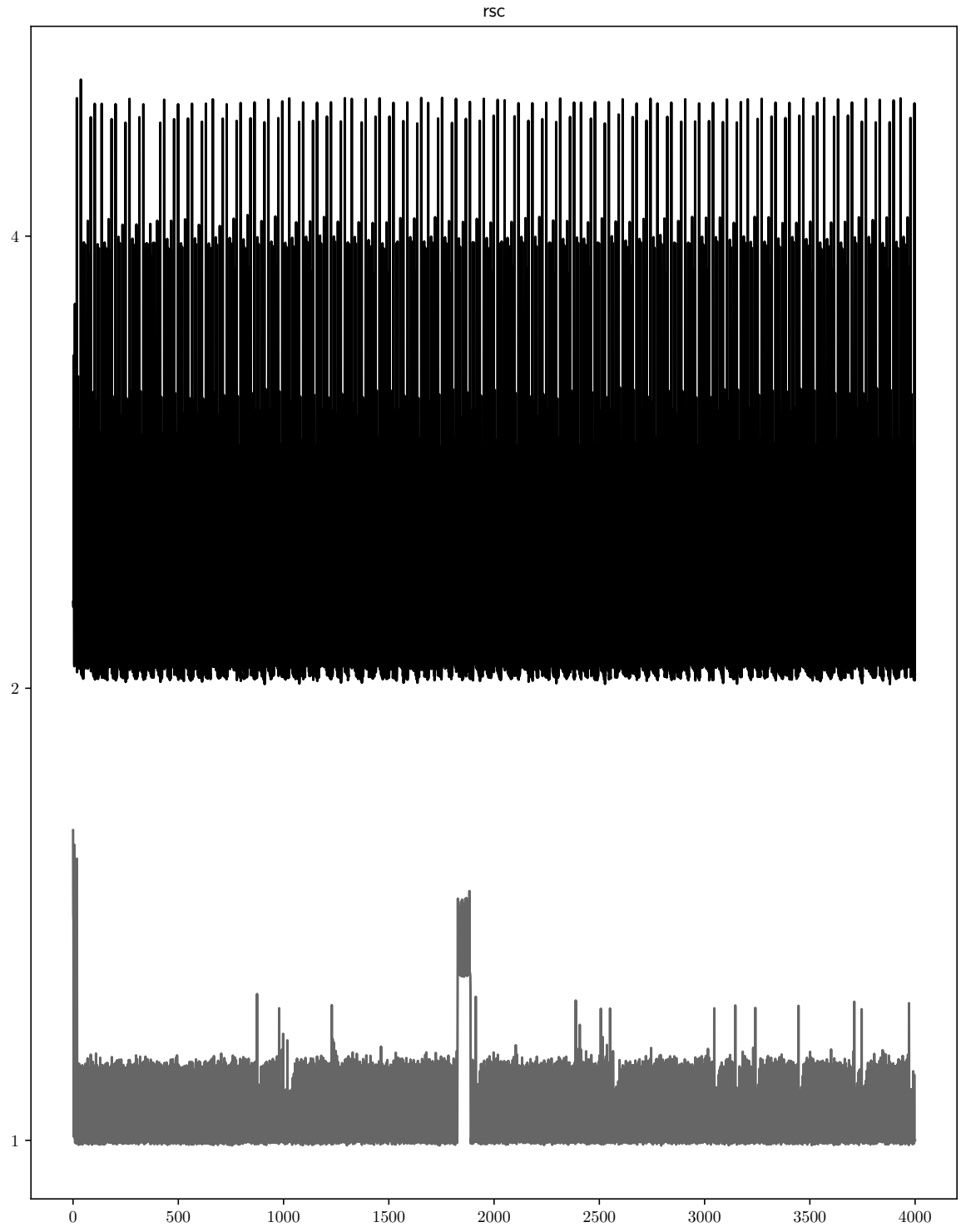


Figure A.26 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the rsc benchmark, less is better.

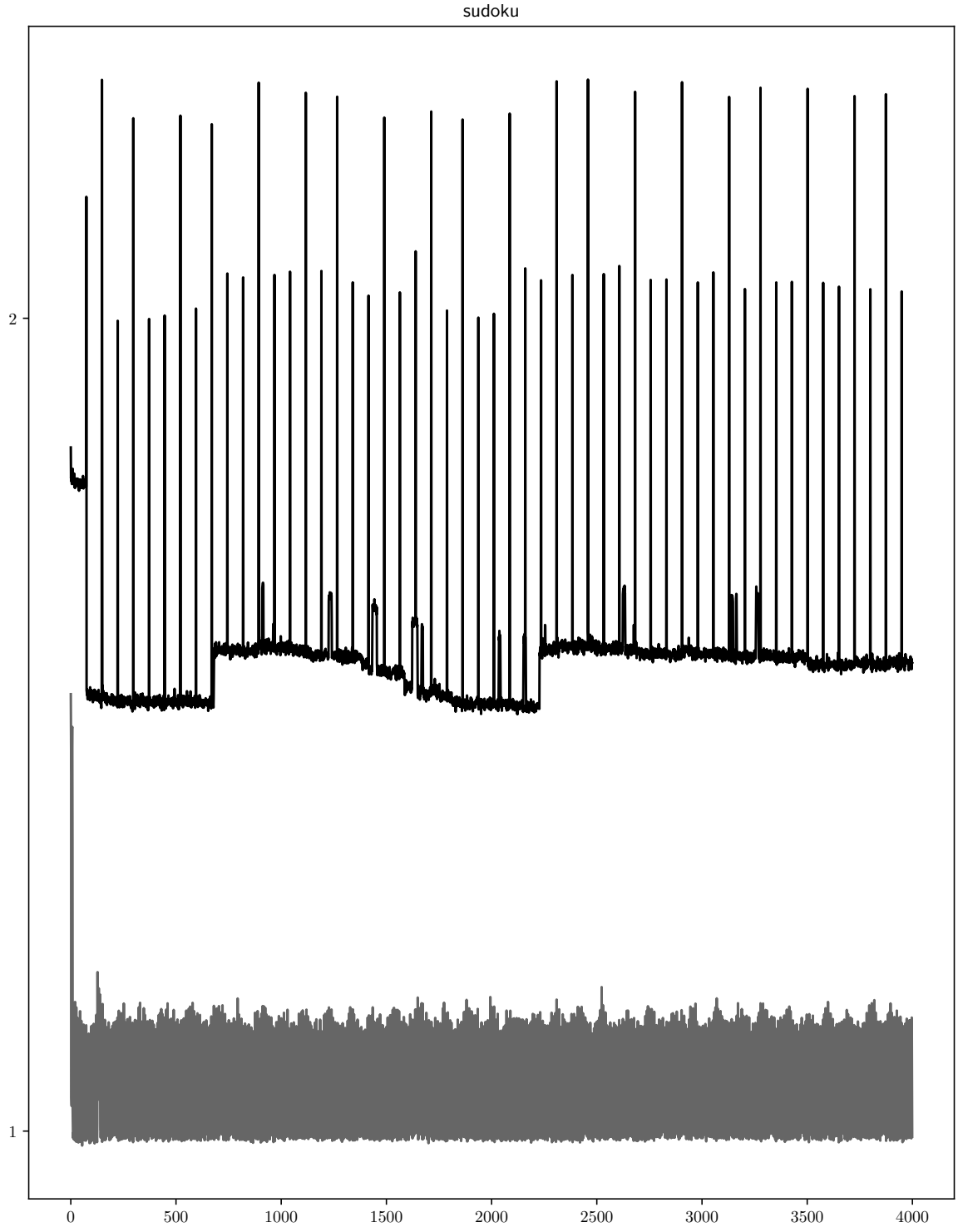


Figure A.27 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the sudoku benchmark, less is better.

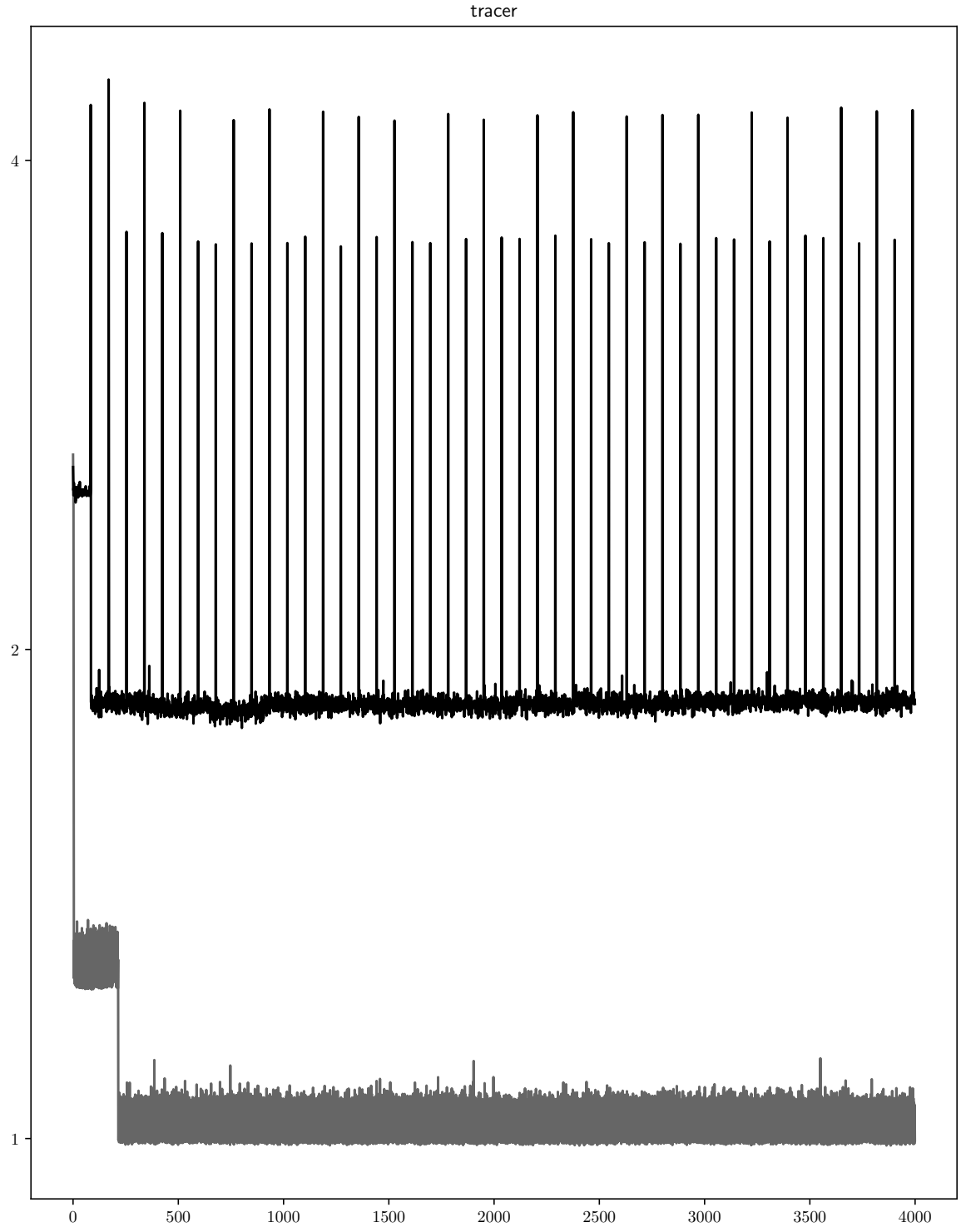


Figure A.28 – Warm-up performance of Interflow with PGO (gray) relative to Native Image (black) on the tracer benchmark, less is better.

Bibliography

[1] (2001). Itanium c++ abi. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Accessed: 2019-09-01.

[2] (2018). Graal native image. https://www.graalvm.org/docs/reference-manual/aot-compilation/. Accessed: 2019-09-01.

[3] (2019). Commix: parallel mark and concurrent sweep gc. https://github.com/scala-native/scala-native/pull/1423. Accessed: 2019-09-01.

[4] (2019). Garbage collection safepoints in llvm. https://llvm.org/docs/Statepoints.html. Accessed: 2019-09-01.

[5] (2019). Mimal ubuntu. https://wiki.ubuntu.com/Minimal. Accessed: 2019-09-01.

[6] (2019). Mostly precise gc. https://github.com/scala-native/scala-native/pull/726. Accessed: 2019-09-01.

[7] (2019). Swift intermediate language. https://github.com/apple/swift/blob/master/docs/SIL.rst. Accessed: 2019-09-01.

[8] Allen, F. E. (1970). Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM.

[9] Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., et al. (2005). The jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417.

[10] Appel, A. W. (2006). *Compiling with continuations*. Cambridge University Press.

[11] Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. (2004a). A survey of adaptive optimization in virtual machines. In *PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION*.

[12] Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. (2004b). A survey of adaptive optimization in virtual machines. In *PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION*.

[13] Bacon, D. F. and Sweeney, P. F. (1996). Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10):324–341.

[14] Bala, V., Duesterwald, E., and Banerjia, S. (2011). Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 46(4):41–52.

[15] Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA. IEEE Computer Society.

[16] Barabash, K., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V., Ossia, Y., Owshanko, A., and Petrank, E. (2005). A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1097–1146.

[17] Barrett, E., Bolz-Tereick, C. F., Killick, R., Knight, V., Mount, S., and Tratt, L. (2017). Virtual machine warmup blows hot and cold. In *OOPSLA*. ACM.

[18] Bartlett, J. F. (1988). Compacting garbage collection with ambiguous roots. *ACM SIGPLAN Lisp Pointers*, 1(6):3–12.

[19] Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*.

[20] Blackburn, S. M., Cheng, P., and McKinley, K. S. (2004a). Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 25–36. ACM.

[21] Blackburn, S. M., Cheng, P., and McKinley, K. S. (2004b). Oil and water? high performance garbage collection in java with mmtk. In *Proceedings. 26th International Conference on Software Engineering*, pages 137–146. IEEE.

[22] Blackburn, S. M. and Hosking, A. L. (2004). Barriers: Friend or foe? In *Proceedings of the 4th international symposium on Memory management*, pages 143–151. ACM.

[23] Blackburn, S. M. and McKinley, K. S. (2008a). Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, volume 43, pages 22–32. ACM.

[24] Blackburn, S. M. and McKinley, K. S. (2008b). Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, volume 43, pages 22–32. ACM.

[25] Blackburn, S. M., Singhai, S., Hertz, M., McKinely, K. S., and Moss, J. E. B. (2001). Pretenuring for java. In *ACM SIGPLAN Notices*, volume 36, pages 342–352. ACM.

[26] Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820.

[27] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. (2009). Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM.

[28] Bothner, P. (2003). Compiling java with gcj. *Linux Journal*, 2003(105):4.

[29] Burmako, E. (2017). Rsc. https://github.com/twitter/rsc. Accessed: 2019-09-01.

[30] Click, C. and Cooper, K. D. (1995). Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196.

[31] Click, C., Tene, G., and Wolf, M. (2005). The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56. ACM.

[32] Cox, R. (2010). Regular expression matching in the wild. https://swtch.com/~rsc/regexp/regexp3.html. Accessed: 2019-09-01.

[33] Dean, J., Grove, D., and Chambers, C. (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer.

[34] Detlefs, D., Flood, C., Heller, S., and Printezis, T. (2004). Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM.

[35] Doeraene, S. (2018). Cross-platform language design in scala. js (keynote). In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, pages 1–1. ACM.

[36] Dragos, I. (2010). Compiling scala for performance.

[37] Driesen, K. and Hölzle, U. (1995). Minimizing row displacement dispatch tables. In *ACM SIGPLAN Notices*, volume 30, pages 141–155. ACM.

[38] Driesen, K. and Hölzle, U. (1996). The direct cost of virtual function calls in c++. In *ACM Sigplan Notices*, volume 31, pages 306–323. ACM.

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

[40] Duesterwald, E. and Bala, V. (2000). Software profiling for hot path prediction: Less is more. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 202–211, New York, NY, USA. ACM.

[41] Geoffray, N., Thomas, G., Lawall, J., Muller, G., and Folliot, B. (2010). Vmkit: a substrate for managed runtime environments. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM.

[42] Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L. (1996). Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138.

[43] Hölzle, U., Chambers, C., and Ungar, D. (1991). Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK. Springer-Verlag.

[44] Huang, X., Blackburn, S. M., McKinley, K. S., Moss, J. E. B., Wang, Z., and Cheng, P. (2004). The garbage collection advantage: improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80.

[45] Jones, R. (1992). Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79.

[46] Kennedy, A. and Syme, D. (2001). Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices*, volume 36, pages 1–12. ACM.

[47] Klonatos, Y., Koch, C., Rompf, T., and Chafi, H. (2014). Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864.

[48] Knuth, D. E. (1984). The complexity of songs. *Communications of the ACM*, 27(4):344–346.

[49] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. (2008). Design of the java hotspot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7.

[50] Lattner, C. (2008). Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2.

[51] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society.

[52] Leopoldseder, D., Schatz, R., Stadler, L., Rigger, M., Würthinger, T., and Mössenböck, H. (2018a). Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 2:1–2:13, New York, NY, USA. ACM.

[53] Leopoldseder, D., Stadler, L., Rigger, M., Würthinger, T., and Mössenböck, H. (2018b). A cost model for a graph-based intermediate-representation in a dynamic compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2018, pages 26–35, New York, NY, USA. ACM.

[54] Leopoldseder, D., Stadler, L., Würthinger, T., Eisl, J., Simon, D., and Mössenböck, H. (2018c). Dominance-based duplication simulation (dbds): Code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 126–137, New York, NY, USA. ACM.

[55] Marr, S., Daloze, B., and Mössenböck, H. (2016). Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131, New York, NY, USA. ACM.

[56] Mohnen, M. (2002). A graph—free approach to data—flow analysis. In *International Conference on Compiler Construction*, pages 46–61. Springer.

[57] Morel, E. and Renvoise, C. (1979). Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103.

[58] Müller, U. (1993). Brainfuck–an eight-instruction turing-complete programming language. http://en.wikipedia.org/wiki/Brainfuck. Accessed: 2018-06-04.

[59] Muth, R. and Debray, S. (1997). Partial inlining. *Unpublished technical summary*.

[60] Nuzman, D., Eres, R., Dyshel, S., Zalmanovici, M., and Castanos, J. (2013). Jit technology with c/c++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):59.

[61] Odersky, M., Blanvillain, O., Liu, F., Biboudis, A., Miller, H., and Stucki, S. (2017). Simplicitly: Foundations and applications of implicit function types. In *45th ACM SIGPLAN Symposium on Principles of Programming Languages*, number CONF.

[62] Odersky, M. and Moors, A. (2009). Fighting bit rot with types (experience report: Scala collections). In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[63] Ossia, Y., Ben-Yitzhak, O., Goft, I., Kolodner, E. K., Leikehman, V., and Owshanko, A. (2002). A parallel, incremental and concurrent gc for servers. In *ACM SIGPLAN Notices*, volume 37, pages 129–140. ACM.

[64] Paleczny, M., Vick, C., and Click, C. (2001). The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, volume 1.

[65] Pall, M. (2008). The luajit project. *Web site: http://luajit. org*.

[66] Petrashko, D., Ureche, V., Lhoták, O., and Odersky, M. (2016). Call graphs for languages with parametric polymorphism. *Acm Sigplan Notices*, 51(10):394–409.

[67] Pettis, K. and Hansen, R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA. ACM.

[68] Prokopec, A., Duboscq, G., Leopoldseder, D., and Würthinger, T. (2019). An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 164–179, Piscataway, NJ, USA. IEEE Press.

[69] Prokopec, A., Leopoldseder, D., Duboscq, G., and Würthinger, T. (2017). Making collection operations optimal with aggressive jit compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 29–40. ACM.

[70] Rastello, F. (2016). *SSA-based Compiler Design*. Springer Publishing Company, Incorporated.

[71] Reames, P. (2017). Falcon: an optimizing java jit. In *LLVM Developers Meeting*, pages 2017–10.

[72] Rompf, T. and Odersky, M. (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM.

[73] Rossberg, A., Titzer, B. L., Haas, A., Schuff, D. L., Gohman, D., Wagner, L., Zakai, A., Bastien, J., and Holman, M. (2018). Bringing the web up to speed with webassembly. *Commun. ACM*, 61(12):107–115.

[74] Shabalin, D. (2019). Scala native benchmarks. https://github.com/scala-native/scala-native-benchmarks. Accessed: 2019-09-01.

[75] Shahriyar, R., Blackburn, S. M., and McKinley, K. S. (2014). Fast conservative garbage collection. In *ACM SIGPLAN Notices*, volume 49, pages 121–139. ACM.

[76] Stadler, L., Würthinger, T., and Mössenböck, H. (2014). Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 165. ACM.

[77] Stallman, R. (2001). Using and porting the gnu compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer.

[78] Stroustrup, B. (2000). *The C++ programming language*. Pearson Education India.

[79] Sujeeth, A., Lee, H., Brown, K., Rompf, T., Chafi, H., Wu, M., Atreya, A., Odersky, M., and Olukotun, K. (2011). Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616.

[80] Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. (2014a). Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134.

[81] Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. (2014b). Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134.

[82] Tene, G., Iyengar, B., and Wolf, M. (2011). C4: The continuously concurrent compacting collector. In *ACM SIGPLAN Notices*, volume 46, pages 79–88. ACM.

[83] Ureche, V., Talau, C., and Odersky, M. (2013). Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. *ACM SIGPLAN Notices*, 48(10):73–92.

[84] Wimmer, C., Jovanovic, V., Eckstein, E., and Würthinger, T. (2017). One compiler: Deoptimization to optimized code. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 55–64, New York, NY, USA. ACM.

[85] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., and Grimmer, M. (2017). Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA. ACM.

[86] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. (2013). One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM.

Denys Shabalin

den.shabalin@gmail.com
M.S. (Ph.D. expected) Computer Science
Lausanne, Switzerland
Ukrainian Citizen

My mission is to advance the state of the art of compiler construction and managed runtimes. In particular, I'm interested in optimizing compilers for modern high-level garbage-collected languages.

The majority of my work has been on the Scala programming language. I designed and implemented Scala Native, an ahead-of-time compiler and managed runtime for Scala based on LLVM, and co-founded Scalameta, infrastructure that enables next-generation developer tools for Scala.

Expertise

- Optimizing compilers
- Managed runtimes

Work History

- École polytechnique fédérale de Lausanne (EPFL)
  Research Assistant
  September 2014 — Present

  My research at EPFL has been centered around Scala Native, an ahead-of-time compiler and managed runtime for the Scala programming language built on top of the LLVM compiler infrastructure. I designed and implemented the project from the ground up, starting from the NIR, Scala Native's intermediate representation.

  My main focus has been on the development of the compiler infrastructure such as the design of the compilation pipeline from Scala to LLVM IR. In particular, I've developed a whole-program flow-sensitive optimizer called Interflow that in combination with LLVM optimizer is 10% faster than the HotSpot JVM on the benchmark suite from our paper. When used to optimize idiomatic Scala code, Interflow produces up to 3x faster results than bare LLVM through a combination of type-driven flow-sensitive devirtualization, partial escape analysis, and inlining.

I coordinated and supervised the design and development of a custom parallel garbage collector that fits into Scala Native runtime based on the Immix garbage collector design. It obtains comparable allocation rates to the parallel collector of the reference JDK implementation while consuming significantly less memory due to its non-copying nature.

I coordinated the open-source development of the project at large. Scala Native has attracted over 80 contributors who have made a major impact on the work towards library compatibility between JVM and Native implementations.

Additionally, as part of my research, I developed Scala Offheap, a library for high-performance off-heap memory management for Scala. It's based upon an efficient implementation of memory pools and completely eliminates the garbage collection cost for applications that are sensitive to the GC pause times.

- Lightbend Inc. (formely Typesafe Inc.)
  Software Engineer Intern
  August 2013 — August 2014

  As an intern at Typesafe Inc, I participated in the development of the official Scala compiler. I designed and implemented quasiquotes — a user-friendly notation for creating and matching abstract syntax trees that made a major impact on how users implement macros in Scala. Quasiquotes are implemented as a compile-time transformation that maps textual snippets of Scala code into lower-level code that constructs or deconstructs its corresponding AST.

  Moreover, I also co-founded Scalameta together with Eugene Burmako. The project has grown to be the foundation for next-generation tools for Scala such as Scalafmt source code formatter, Scalafix migration tool, and Metals language server, as championed by Ólafur Páll Geirsson. I co-authored the initial design of the syntactic APIs (AST and token level introspection and rewriting) and implemented a Scala parser that works on top of those APIs.

- Fotobooka.com
  Software Engineer Contractor
  May 2011 — August 2012

  Fotobooka.com is a photo album printing service that lets users compose their custom albums using WYSIWYG software. During my work at the company, I implemented a desktop client software for using a combination of Python and Qt technologies.

  I also participated in the design and implementation of the backend software that

integrated the user-provided album designs into the publishers printing workflow. This includes the devops infrastructure for the deployment of the underlying web service.

Education

- École polytechnique fédérale de Lausanne (EPFL)
  Ph.D. Computer Science: September 2014 — October 2019 (expected)
  M.S. Computer Science: September 2012 — June 2014

- National University of Kyiv-Mohyla Academy
  B.S. Applied Mathematics: September 2008 — June 2012

Written Work and Publications

- "Interflow: interprocedural flow-sensitive type inference and method duplication"
  Denys Shabalin, Martin Odersky. Scala Symposium 2018, St. Louis, MO, 2018.

- "Region-based off-heap memory for Scala"
  Denys Shabalin, Martin Odersky, Technical Report, 2015.

- "Hygiene for Scala"
  Denys Shabalin, Jason Zaugg, Martin Odersky. Master Thesis, 2014.

- "Quasiquotes for Scala"
  Denys Shabalin, Eugene Burmako, Martin Odersky. Technical Report, 2013.

Tech Talks

- "Interflow"
  Scala Symposium 2018, St Louis, MO, September 2018.

- "Scala Native"
  SF Scala meetup talk, San Francisco, CA, July 2017.

- "Fast startup & low latency: pick two"
  Tech talk at Scala Days 2017, Copenhagen, Denmark, June 2017.

- "Coding up your first game in Scala Native"
  Live coding tech demo at Scala Matsuri 2017, Tokyo, Japan, March 2017.

- "Managing Your Resources"
  Tech talk at Scala World 2016, Lake District, UK, 2016.

- "Scala goes Native"
  Tech talk at Scala Days 2016, New York City, NY, May 2016.

- "Type-safe off-heap memory for Scala"
  Tech talk at Scala Days 2015, Amsterdam, Netherlands, June 2015.

- "Quote or be quoted"
  Tech talk at Scala Days 2014, Berlin, Germany, June 2014.