Diplomarbeit

Adaptive Inlining and On-Stack Replacement in a Java Virtual Machine

ausgeführt am

Institut für Computersprachen
Arbeitsbereich für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.tech. Andreas Krall

durch

Edwin Steiner
Hietzinger-Kai 127/2/38
1130 Wien

6. Februar 2007

Abstract

Method inlining is a well-known and effective optimization technique for object-oriented programs. In the context of dynamic compilation, method inlining can be used as an adaptive optimization in order to eliminate the overhead of frequently executed calls.

This work presents an implementation of method inlining in CACAO, a just-in-time compiler for Java. On-stack replacement is used for installing optimized code and for deoptimizing code when optimistic assumptions of the optimizer are broken by dynamic class loading.

Three inlining heuristics are compared using empirical results from a set of benchmark programs. A type-based algorithm for the elimination of local subroutines is described.

Kurzfassung

Methoden-Inlining ist eine bekannte und wirkungsvolle Technik für die Optimierung objekt-orientierter Programme. Im Zusammenhang mit dynamischer Übersetzung kann Methoden-Inlining als adaptive Optimierung verwendet werden, um den Mehraufwand von häufig ausgeführten Methodenaufrufen zu vermeiden.

Diese Arbeit stellt eine Umsetzung von Methoden-Inlining in CACAO vor, einem *just-in-time* Übersetzer für Java. Das Ersetzen von Methoden im Aufrufstapel ermöglicht die Installation von optimiertem Maschinencode und das Rückgängigmachen von Optimierungen, wenn optimistische Annahmen des Optimierers durch dynamisches Nachladen von Klassen ungültig werden.

Drei Heuristiken für Methoden-Inlining werden anhand empirischer Messungen an einem Satz von Benchmarkprogrammen verglichen. Ein typenbasierter Algorithmus zum Entfernen lokaler Unterprogramme wird beschrieben.

Acknowledgements

First, I want to thank my family for their support throughout my education.

I also thank Prof. Andreas Krall, for giving me the opportunity to work on the CACAO project.

Special thanks go to the members of the CACAO Team, in particular to Dipl.-Ing. Christian "`TWISTI`" Thalinger, for a long time of joyful cooperation.

Contents

Chapter 1

Introduction

Modern programming languages offer sophisticated abstraction mechanisms to the programmer. Important examples are subroutines, classes, virtual methods, and synchronization constructs. The challenge for a language implementation is to support these abstractions with as little overhead as possible, so programmers can make unrestricted use of language features and maintain good programming style, instead of caring about implementation details.

In parallel to the evolution of programming languages, the underlying run-time systems have to provide more advanced features. The Java [21] run-time system—implemented in form of a Java virtual machine [27]—for example, provides exceptions, automatic garbage collection, multiple threads of execution, synchronization mechanisms, and dynamic linking. In such an environment, classical program optimizations often have to be modified in order to still be applicable and effective.

1.1 Inlining

Each abstraction mechanism introduces new challenges for the language implementation. Subroutines, for example, allow the programmer to factor programs into small units of functionality. Object-oriented programming styles in particular favor lots of very small subroutines (called *methods* in this context) and frequent, deeply nested calls. The language implementation does not only have to minimize the call overhead. Subroutine boundaries severely limit the scope available to optimizations, decreasing their benefits.

An important technique for addressing these problems is *inlining*. Inlining refers to replacing subroutine calls with a modified version of the body of the called subroutine. This effectively reverts the abstraction introduced by the programmer, eliminating call overhead, and providing larger scopes for subsequent optimization. There are, however, several issues which complicate inlining: First, inlining is not without costs. Excessive inlining can

greatly increase compiled code size, deteriorating compile time and runtime performance. Thus, the need arises to make good *inlining decisions*, selecting only the most beneficial sites for inlining.

Another complication is caused by *polymorphic calls* that must be dispatched at runtime. The receivers of such calls cannot be determined at compile time, making straight-forward inlining impossible.

Finally, recursive calls obviously cannot be inlined in a simple manner.

1.2 Adaptive Optimization

Heterogeneous computing platforms make it appealing to distribute software in portable source code or intermediate (bytecode) form. This, however, requires time-consuming compilation on each platform, or interpretation at the cost of performance. Techniques like just-in-time compilation are used to reconcile the conflicting aims of responsiveness and performance.

In the presence of dynamic linking, parts of the program code may only become available during execution of the program. This precludes the use of classic whole-program optimizations. However, it is possible to modify optimizations such that they act upon *preliminary* results obtained at runtime. In order to guarantee correctness, such speculative optimization must be supplemented by mechanisms that track the assumptions made during optimization and take proper measures as soon as any assumption becomes invalid—for example due to new code being dynamically loaded.

Compiling parts of the program at runtime opens new possibilities for optimization: *Adaptive optimization* refers to analyzing the behavior of the running program and optimizing the most frequently executed code based on the collected data. In this context, *recompilation* is used to create more efficient code that replaces already compiled code, as soon as enough profile data is available.

1.3 Replacement

Replacing program code during runtime poses great challenges for a virtual machine. Particular difficulties arise with *on-stack replacement*, i.e. replacing the code of methods that are currently activated ("on the stack"). Nevertheless, on-stack replacement is an attractive technique. It allows timely replacement of methods that execute for a very long time. Additionally it enables, for example, speculative optimizations like inlining currently-monomorphic calls without paying the runtime cost of guard code to protect the optimized call sites.

1.4 The CACAO Virtual Machine

The techniques described in this work have been implemented in CACAO [25], a Java Virtual Machine providing just-in-time compilation for several processor architectures. In addition to code generators for i386, x86_64, alpha, MIPS, MIPS32, PowerPC, PowerPC 64[1], ARM, and SPARC 64[1], CACAO provides bytecode verification, a full implementation of dynamic class loading and linking, and exact garbage collection[1].

1.5 Overview

The remainder of this work is organized as follows: Chapter 2 summarizes related work on inlining, on-stack replacement, local subroutines, and adaptive optimization in general. Chapter 3 describes the adaptive optimization framework implemented in the CACAO virtual machine. Chapter 4 first gives an overview of method inlining and then deals with specific problems and how they were solved in CACAO. Chapter 5 discusses local subroutines and presents an approach for eliminating them. Chapter 6 deals with code replacement in a virtual machine using recompilation, including a detailed description of on-stack replacement as implemented in CACAO. Chapter 7 briefly discusses how CACAO's inlining and on-stack replacement code has been tested. Chapter 8 presents empirical results that were obtained with the CACAO implementation. Finally, Chapter 9 summarizes the results.

[1] currently in development parallel to this work

Chapter 2

Related Work

A large amount of research has been done in the field of adaptive optimization in general and into effective inlining in particular. This chapter discusses some related work in this area.

2.1 Method Inlining

Inlining (also known as *procedure integration* and *unfolding*) has been a well-known program transformation for over three decades [2]. Consequently there is a large body of work about its effectiveness, implementation and consequences. The following sections give an overview of related work on selected topics.

2.1.1 Effectiveness of Inlining

Inlining can be counted among the most effective optimizing program transformations for a variety of programming languages, with examples of execution time improvements of 5 to 28% [30] for CLU[1], 15% [14] for C code, 24% [6] for intermediate code, and 10 to 44% [23] or up to 40% [4] for Java programs.

Scheifler [30] gave an early analysis of the effectiveness of inlining substitution for structured programming languages. Scheifler already observed that most of the benefit from inlining does not come from the elimination of call overhead, but from opening more possibilities for down-stream optimizations—a statement which is still valid today.

Inlining can both significantly decrease and increase the execution time of programs, depending on the program and the selection of inlining sites. Most papers assume that the degradation of performance observed in some cases is due to the expansion of compiled code size caused by inlining. However, a detailed analysis by Davidson and Holler [14] showed that this effect

[1] a structured programming language

is generally overestimated. In some cases code size was increased by a factor of 3 to 8 without any impact on execution speed. On average inlining also slightly improved the cache performance of programs. Davidson and Holler demonstrated that the effects of inlining on register save and restore overhead and on the quality of register allocation are responsible for most of the performance deterioration.

2.1.2 Inlining Heuristics

Making optimal inlining decisions, even if the whole program is known to the compiler, is an intractable problem [30]. On the other hand, bad inlining decisions can have significant costs in terms of compiled code expansion and increased compile time, while delivering little or no benefit. Consequently there has been a lot of research into *inlining heuristics* that try to select the most profitable calls to inline while limiting code expansion.

Scheifler [30] showed that inlining can be viewed as a variant of the NP-complete KNAPSACK problem [1]. He proposed a greedy algorithm that selects the inlining option with the highest ratio of expected executions to code size increase. The algorithm repeats until code expansion reaches a given threshold.

Arnold, Fink, Sarkar, and Sweeney conducted a comparative study [4] of inlining heuristics. They demonstrated that the effectiveness of inlining rises with the precision of available profile data. In particular, heuristics based on a dynamic call graph with edge weights proved to outperform all algorithms using coarser data, even if given much narrower limits for code expansion.

Ishizaki et al. [23] reported that inlining only very small methods gave performance improvements to within 15% of the peak performance, while increasing compilation time by only 6% on average. On the other hand, in order to obtain the peak performance, an increase of compile time by up to 50% had to be accepted.

As Davidson and Holler [14] demonstrated, code size should not be the only concern of inlining heuristics. For example, inlining a rarely called method in a method that is frequently entered can significantly increase the number of register saves and restores performed, thereby deteriorating performance. This problem could be addressed by better placement of save and restore instructions [11]. Otherwise[2], inlining should be limited to frequently traversed edges of the call graph, even if the increase in code size would be small.

Given a set of inlining heuristics, an important problem is how to choose the parameters of these heuristics in order to obtain the best performance on a given target architecture. Cavazos and O'Boyle [8] developed a genetic

[2]The CACAO compiler always places register save and restore instructions in the method prolog and epilog, respectively.

algorithm for off-line tuning that can automatically find parameters yielding good performance for a given set of benchmark programs.

2.1.3 Devirtualization

Object-oriented languages typically make use of dynamically dispatched *virtual methods* in order to support polymorphism. Virtual methods present great challenges to an optimizing compiler. As the receiver of a virtual call is not known during static analysis, all possible receivers must be considered. This decreases the precision of static analysis and makes inlining virtual calls impossible. However, not all call sites of virtual methods are truly polymorphic. The purpose of *devirtualization* is to identify virtual call sites that can be proven to be monomorphic and turn them into statically bound calls. Even if the call site is polymorphic, i.e. there are multiple receiver types, there may still be only a single target method that can be statically bound.

Dean, Grove, and Chambers proposed *class hierarchy analysis* [16] (CHA) as a means to limit the set of potential receivers of virtual calls.

Zaks, Feldman, and Aizikowitz demonstrated [32] how Java's security mechanism of *sealed packages* can be used to devirtualize calls in the presence of dynamic loading.

Lee et al. reported [26] that in the Java programs of the SPECjvm98 benchmark suite, about 85% of virtual calls are monomorphic and about 90% have a single target method.

2.1.4 Guarded Inlining

Guarded inlining techniques can be used to inline methods that cannot be proven to be monomorphic. In this case, test code is generated before the inlined code to check the receiver of the call. There are several interpretations of guarded inlining: Dean [17] proposed *exhaustive class testing* for the case that all possible receivers can be inlined. He described singleton class tests and subclass tests that can be used to classify receivers.

Detlefs and Agesen proposed the *method test* [18] as a refinement of the class test for guarded inlining.

Arnold and Ryder introduced *thin guards* [5] in order to further reduce the cost of guarded inlining.

2.1.5 Preexistence-based Inlining

Detlefs and Agesen suggested *preexistence-based inlining* [18] as a way to perform direct speculative inlining without the need for on-stack replacement. The authors report that in a set of benchmarks 40% of all virtual call sites could be proven to have preexisting receivers using a simple *invariant*

arguments analysis. Their results also showed that direct inlining can improve performance substantially compared to guarded inlining (up to over 38% for the SPECjvm98 `mtrt` benchmark).

2.1.6 Inlining Trials

One of the problems of heuristically guided inlining is that the resulting code may be less efficient than the original, because the applied heuristics underestimated the resulting code size, or overestimated the benefits of following optimizations. On the other hand profitable inlining opportunities may be wasted because the down-stream benefits are underestimated.

Dean and Chambers [15] propose *inlining trials* as a remedy for this problem. With inlining trials, inlining is experimentally performed and the inlined code is optimized in the context of its caller. Only then does the compiler assess costs and benefits and decide whether to keep the inlined version, or discard it. The results of inlining trials are stored in a persistent database, so future compilations can reuse the data to guide inlining decisions. A downside of inlining trials is that they require rather complex monitoring of the information used for optimizing the inlined code. The results obtained by Dean and Chambers suggest that inlining trials are more effective for saving compilation time and code size rather than improving execution time.

2.1.7 Unexpected Side Effects of Inlining

Cooper, Hall, and Torczon point out [13] that inlining can have unexpected side effects that deteriorate the performance of the compiled code, because the inlining transformation looses information about the original shape of the program. However, the example they provide depends on an idiosyncrasy of the FORTRAN 77 standard that has no counterpart in the Java language. Generally the problem is that information about restrictions which the language imposes at subroutine boundaries are lost in the inlining transformation and thus cannot be used by subsequent optimizations. To the author's knowledge this problem does not occur with Java bytecode, at least not beyond information that standard analysis techniques can recover. In particular, bytecode verification is performed before method inlining, so all static guarantees [27] on Java bytecode can be established. However, some difficulties that are outlined in later sections of this work (e.g. the effects of inlining on a naïve register allocator and the need for explicit null pointer checks) can be regarded as instances of this fundamental issue.

2.2 Local Subroutines

The Java Virtual Machine [27] supports *local subroutines* in order to reuse code within a method. Local subroutines increase the complexity of control-flow and data-flow analysis of the bytecode, and pose challenges for the bytecode verifier.

Freund [20] has given statistical evidence that local subroutines are not very effective in reducing the size of Java bytecode. For example, he reports that the savings by local subroutines in the JDK 1.1.5 amount to only 0.02% of total bytecode size.

Coglio developed a type-based technique [12] for verifying local subroutines. This technique has been implemented in the CACAO verifier. It is also used as the basis for local subroutine elimination in CACAO.

2.3 Adaptive Optimization Frameworks

Several frameworks for dynamic compilation and adaptive optimization of Java code have been developed. These frameworks can be coarsely divided in two classes: on the one hand there are *compile-only* frameworks which use a fast *baseline compiler* for generating the initial code for all invoked methods, and one or more *optimizing compilers* for *recompiling* frequently executed code. On the other hand there are *mixed-mode* systems that employ an interpreter to execute most of the program and only compile hot methods.

The Jalapeño Virtual Machine [3] developed by Burke et al. was the first [7] *compile-only* system for dynamic optimizing compilation of Java code.

The Java HotSpot Server Compiler [28] uses a mixed-mode approach for executing Java code. HotSpot employs method-entry and backward-branch counters to select hot methods for compilation. On-stack replacement is used for deoptimization and for replacing long-running methods (see the following section).

Suganuma et al. [31] developed another mixed-mode framework that provides a three-level optimizing compiler, a lightweight continuous sampling profiler, and an instrumenting profiler. Their instrumenting profiler allows dynamic installation and de-installation of profiling code at method entries by code patching.

On a general level, all of these frameworks have some components in common: Some kind of profiling system to collect data, a database to store collected data and optimization decisions, a controller, the actual compiler(s), and a mechanism for installing newly compiled code and (optionally) uninstalling obsolete code.

2.4 On-Stack Replacement

On-stack replacement was first described by Chambers and Ungar in the context of deferred compilation in the Self-91 compiler [10]. Hölzle, Chambers and Ungar subsequently used on-stack replacement for debugging optimized code via deoptimization [22]. The *interrupt points* defined by the authors are analogous to the replacement points used in this paper.

In his PhD Thesis [9] Chambers described *scope descriptions* and *byte code mappings* created by the SELF Compiler in order to facilitate deoptimization. He also presented *dependency links* as a means to record assumptions and perform selective code invalidation.

The Java HotSpot^{TM} Server Compiler [28] uses on-stack replacement to convert natively compiled frames to interpreter frames when either an uncommon trap is reached, or class loading invalidates assumptions made during compilation. In HotSpot, on-stack replacement is also used for replacing methods with long-running loops after recompilation. In this case HotSpot compiles the method with a special entry point at the target of a backward branch, so control can be transfered from the interpreter to native code during execution of the loop.

Fink and Qian implemented and evaluated [19] on-stack replacement in the Jikes RVM. Their design compiles a specialized version of the method for each activation that is replaced. Each version has a specialized prologue prepended to the original bytecode that sets up local variables and the stack and then jumps to the current program counter. An advantage of this scheme is that it requires minimal changes to the underlying compilers and that it may provide additional opportunities for optimization. On the other hand more compiled code has to be generated.

Chapter 3

Adaptive Optimization Framework

Adaptive optimization refers to the application of optimization techniques at runtime by monitoring the behavior of the running program and using the collected data to guide optimization decisions. Several parts of a virtual machine have to cooperate to make adaptive optimization possible. This chapter briefly describes the modules that are responsible for profiling, recompilation, and optimization in the CACAO virtual machine. It also gives an overview of the intermediate representation used in CACAO and points out how the intermediate representation accommodates particular needs of the inliner and the replacement mechanism.

3.1 Modules

In order to provide adaptive optimizations, several modules of the virtual machine must work together. Figure 1 shows the most important modules and how they interface with each other.

JIT compiler The module responsible for compiling bytecode to machine code. See section 3.2 for the internal structure of the compiler.

code repository This module manages the generated machine code, including the invalidation of obsolete code.

method database This is where properties of methods and assumptions about methods are kept.

replacement mechanism This module performs replacement of old versions of compiled code with new versions, including on-stack replacement. The replacement module also provides services to the garbage collector, like setting traps, reading the source state, and writing back a modified source state.

linker The linker determines object layout, performs method overwriting, and builds the virtual method and interface tables.

profiler The profiler conducts measurements in order to find hot spots in the program, and for providing other modules with data to base optimization decisions on.

inliner This is a separate pass in the compiler that performs the inlining transformation. The inliner is depicted as a module in order to emphasize its separation from other parts of the compiler and to clearly show its interaction with the method database. (Other compiler passes are also modules in their own right, but were left out from Figure 1 to avoid cluttering the diagram.)

GC The exact garbage collector interfaces with the replacement mechanism for setting GC traps, finding live objects in the source state, and redirecting references after compaction.

architecture layer This module bundles architecture-dependent functions needed by the replacement mechanism.

3.2 Compiler Passes

In CACAO the baseline compiler and the optimizing compiler operate on the same intermediate representation. The compilers also share code for common passes like bytecode parsing and stack analysis. Compared to the baseline compiler, the optimizing compiler performs additional optimization passes, including inlining and SSA-based optimizations, and it uses a more sophisticated register allocator[1]. Figure 2 shows the sequence of passes for the baseline and optimizing paths.

parser The parser transforms the Java bytecode into the more stream-lined intermediate instruction format. Other functions of the parser are: finding basic block boundaries, creating a bytecode-PC-to-basic-block mapping, and calculating the local variable renaming.

stack analysis This pass performs an abstract interpretation of the intermediate code in order to transform the stack-based data-flow into a data-flow using temporary variables. Stack analysis also does local subroutine elimination and resolving of branch targets to basic block

[1]At the time of this writing the linear scan register allocator was not available in CACAO, yet, so the result of the inlining transformation was passed on to `simplereg` as depicted by an extra arrow in Figure 2.

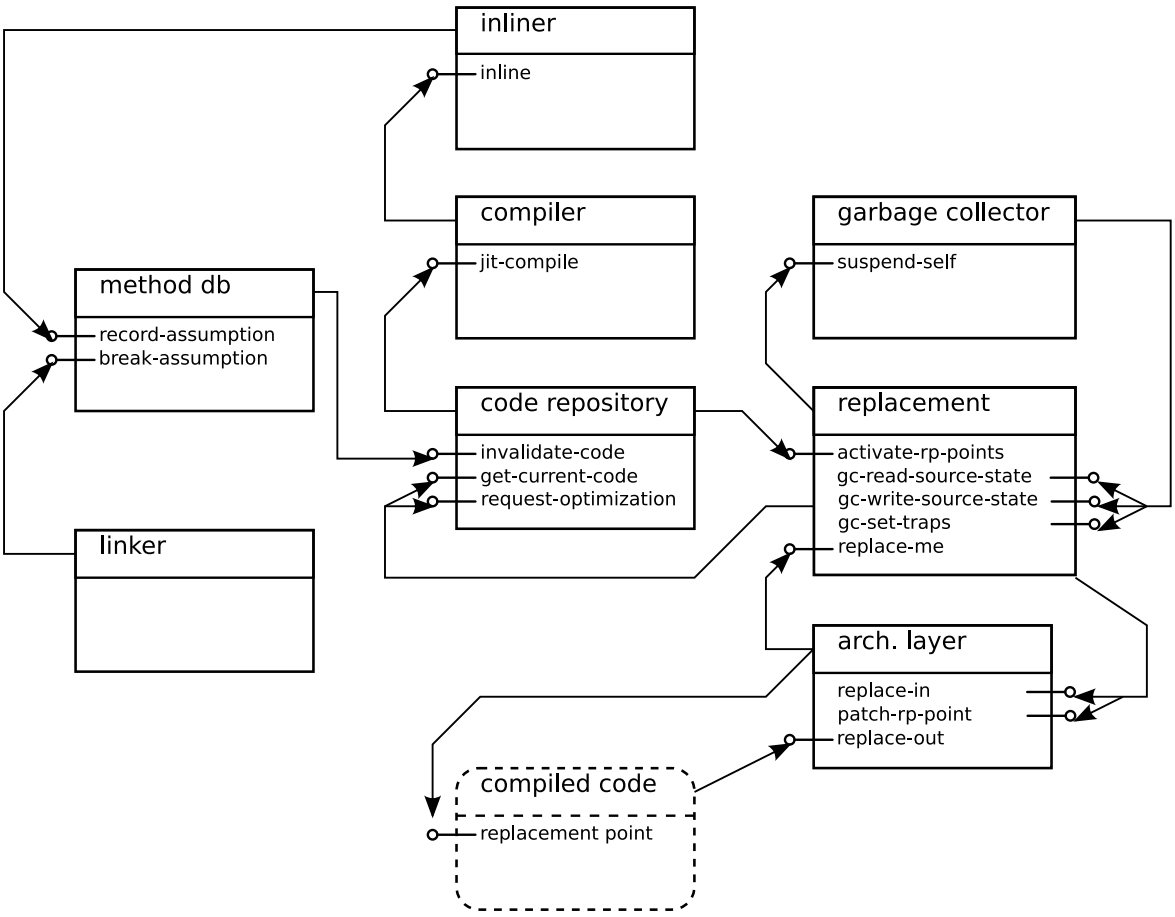

Figure 1: Modules of the adaptive optimization framework

references. In order to generate rather good code even in the baseline compiler, stack analysis also performs some fast peep-hole optimizations for instructions with constant operands, pre-colors argument registers, and performs some basic copy elimination for local variables [24].

type checker The type checker performs type inference to obtain precise type information for intermediate variables. Compatibility of types is checked as a required part of Java bytecode verification [27]. The type information derived in this step can be used by subsequent optimizations.

inlining This pass performs the inlining transformation.

SSA transformation This pass splits local variables in order to transform the intermediate representation into SSA form.

optimization Various optimizations can be performed on the IR. Some optimizations depend on the intermediate representation in SSA form:

- copy propagation
- dead code elimination

Other optimizations can be used independently from SSA form:

- if-conversion
- basic block reordering

register allocation This pass assigns hardware registers or stack locations to the intermediate variables. CACAO provides two register allocators: **simplereg**, a fast and simple local register allocator [24], and **LSRA**, a linear scan register allocator [29].

replacement point (RP) generation The locations of replacement points are determined and the necessary data structures are created.

code generator The code generator finally emits machine code for the compilation unit.

3.3 Adaptive Recompilation

As the optimizing compiler has higher compile time and can produce larger code (by inlining) than the baseline compiler, it should only be used for code that is executed frequently enough to amortize these costs. CACAO uses instrumentation of the code generated by the baseline compiler in order to select methods for which recompilation is likely to be profitable.

The sequence of states and transitions that a method can go through in CACAO is as follows:

baseline + counters The baseline compiler generates code for the method and inserts countdown traps (see Section 6.5.7) in order to trigger recompilation after a certain number of method entries or loop iterations. The majority of methods does never reach this threshold (see Chapter 8).

full instrumentation[2] When a method triggers a countdown trap, it is recompiled with instrumentation code for creating a dynamic profile of the method, including execution counts for each basic block. Countdown traps are used to limit the time the code runs with full instrumentation.

[2]This state is currently not used in CACAO but will likely be added in future versions.

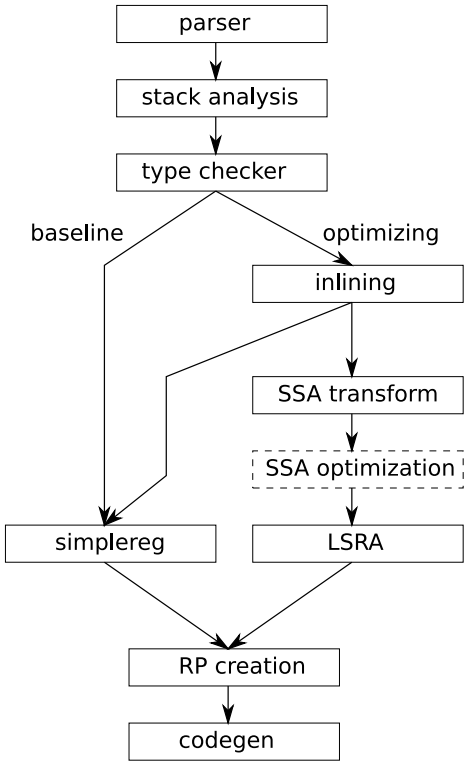

Figure 2: Compiler passes

optimization When the instrumented code triggers recompilation, it is recompiled by the optimizing compiler. The optimized code contains no intrusive instrumentation.

sampling In future versions of CACAO it is planned that a sampling profiler will continuously inspect which methods are running. The sampling profiler can detect optimized methods that contribute significantly to execution time and select them for repeated recompilation at higher optimization levels.

3.4 Intermediate Representation

The JIT compiler uses a unified data structure to transfer the intermediate representation from one compiler pass to the next.

The intermediate representation has the following main components:

- the variable array,
- the basic block list.

The *variable array* keeps the type and other information for each variable. Note that these internal variables do not have a one-to-one correspondence with bytecode variables. There are two kinds of variables:

- local variables—these can be referred to by all basic blocks,
- block variables—these are local to the basic block that defines them. With respect to block variables the intermediate representation is in SSA form.

For each variable the IR stores:

- the basic type—`int`, `long`, `float`, `double`, *returnAddress*, or reference,
- at most one of:
  - allocation info (register number or stack offset),
  - constant value (e.g. for *returnAddress*es after local subroutine elimination.
- a flag indicating whether the variable must be saved across calls.

Basic blocks in the IR are not terminated by method calls or PEIs. Each basic block in the IR contains:

invars An array of block variable indices that are live-in at the basic block start.

outvars An array of block variable indices that are live-out at the basic block end.

instructions The array of intermediate instructions making up the body of the block.

The intermediate instructions are stored in an extended variant of quadruple code. Each intermediate instruction contains:

- the opcode,
- source operands, which may be:
  - up to three variable indices,
  - an argument count and a pointer to an array of variable indices, or
  - a single variable index and a `long`-size constant.
- special constant operands: Free source operand slots are used to hold special constant operands, e.g. the field reference of a `GETFIELD` instruction.

- a destination operand, which may be:
  - a variable index, or
  - a branch target, or
  - a pointer to an array of branch targets (for `TABLESWITCH`), or
  - a pointer to an array of value/branch target pairs (for `LOOKUPSWITCH`).
- flag bits
- the source line number

3.4.1 Liveness Information

Some passes of the compiler require information on live variables. In particular the creation of replacement points (see Section 6.5.5) needs to know the set of live variables at certain points in the program, namely:

- at basic block starts,
- at call sites,
- at other instructions that can indirectly cause calls into native or Java code, for example through patchers[3].

At basic block starts the set of live variables is taken to be the union of all local variables and the *invars* of the basic block. As soon as a more sophisticated liveness analysis for locals is available, the *invars* array can be re-defined to not only refer to block variables but to all live-in variables at basic block start.

At call sites all local variables and the arguments of the call are considered live. In addition each call instruction keeps a list of block variables that live through the call. (Initially these are the variables for operand stack slots "below" the argument slots.)

3.4.2 Local Variables Mapping

In order to achieve unified treatment of local and stack variables, CACAO renames the untyped local variables of Java bytecode to typed intermediate variables. The optimizing compiler performs further renaming and splitting of local variables to transform code into SSA form. Still, on-stack replacement must be able to map variables in a way that is completely independent from any optimizations and code transformations (see Section 6.5.2).

[3]In CACAO, code patching is used to handle unresolved method/field/class references. In such cases a call to a *patcher* function is installed at the unresolved instruction. The patcher performs the resolving of the reference and replaces the call to itself with the final code for the resolved instruction.

To provide a stable mapping of the intermediate variables back to the variables of the bytecode, the IR is augmented with a *javalocals* array for each basic block start. This array is indexed by the Java bytecode local variable index and contains the index of the intermediate variable that represents this bytecode variable at the given program point.

3.4.3 Invariant Instruction ID

Replacement requires a reliable way of identifying program points within a method. Thus the intermediate representation must provide a mapping of intermediate instructions back to the original bytecode positions.[4] More specifically, what we need is an identifier for program locations that is invariant with regard to any optimizations performed on the intermediate representation, and that is unique for any instruction that can become the site of a replacement point (see 6.5.5).

CACAO currently provides such an identifier as follows: During parsing of the bytecode, the intermediate instructions are numbered sequentially. As parsing does not perform any optimizations, this numbering is invariant for a given method and isomorphic to the bytecode position. All subsequent program transformations preserve this identifier when transforming an instruction.

The plan for future development is to label immediate instructions with the bytecode index of the original bytecode instruction, so a general mapping from intermediate instructions to bytecode indices—and consequently a mapping from machine code positions to bytecode indices and source lines—can be obtained.

3.4.4 Comparison of Quadruple Code and Stack Representation

Previous versions of CACAO used an IR based on a representation of the operand stack, in contrast to the new IR using explicit variable operands. For code generation both representations are largely equivalent, as the stack-based IR can be regarded as a kind of quadruple code with implicit source operands. There are, however, some trade-offs in choosing one representation over the other. The stack-based IR has some advantages:

- The top-of-stack—and thus the whole operand stack—is known at each intermediate instruction, so the set of live variables[5] can be obtained at each instruction without augmenting the IR in any way.

[4] As explained in the rest of the paragraph, for replacement the mapping need not really map to meaningful bytecode indices, although such a mapping is useful in other respects, for example when implementing JVMTI functionality.

[5] Actually a conservative approximation thereof is obtained, unless dead stores are detected.

- Instructions taking a variable number of arguments can be expressed naturally.

On the other hand, a stack-based IR has significant disadvantages, which ultimately led to the explicit quadruple code being chosen:

- Local variables and stack variables have fundamentally different representation, so their treatment cannot be unified.
- Instructions which copy or reorder stack slots (e.g. `SWAP` and `DUP_X1`) are very cumbersome to represent. In particular there is no such thing as a generic `MOVE` instruction, which is often needed in program transformations.
- The stack model binds instructions very tightly to each other, even if they have no data-dependencies. Re-arranging instructions, for example during inlining, requires a highly complex relocation of the stack elements.

These points were deemed significant reasons for implementing a quadruple code IR which provides:

- unified representation of local and stack variables as indices into a single variable array,
- generic `MOVE` and `COPY` instructions[6] which enable the elimination of all complex `SWAP` and `DUP`-variants in an early *stack analysis* pass (see Section 3.2),
- simple cloning and relocation of instructions.

[6] `MOVE` and `COPY` are differentiated by the **simplereg** allocator: `MOVE` ends the life range of its source argument, `COPY` does not. For all other purposes these instructions are identical.

Chapter 4

Method Inlining

This chapter describes the implementation of method inlining in CACAO's optimizing compiler.

4.1 Placement of the Inlining Transformation

Regarding the placement of the inlining transformation with respect to other passes and optimizations of the compiler, there are two basic choices:

1. Performing inlining as part of the bytecode reading stage,
2. performing inlining on the intermediate representation at a later stage.

Both approaches have their advantages. Inlining as part of the bytecode reading stage has the advantage that several analysis steps in the following stages can be performed naturally on the inlined code, so there is no need to revise the results later. Examples are:

- determining basic block boundaries,
- deciding whether the method under compilation is a leaf method,
- stack analysis (stack slots living through the call site being inlined are handled naturally).

On the other hand implications of inlining can severely complicate the stages following bytecode reading, especially in a VM like CACAO that performs verification on the intermediate representation.

Making inlining a separate pass on the IR can thus have significant advantages:

- separation of concerns—all code required for method inlining resides in a single pass,

- other passes of the compiler can be easily shared between the baseline compiler and the optimizing compiler,
- inlining can potentially use more information for inlining decisions and for generating good inlined code.

As previous attempts to implement inlining at the bytecode reading stage proved to be excessively intrusive in CACAO, and sharing code between baseline and optimizing compiler was an important criterion, the choice was made to separate out the inlining mechanism to an independent pass over the IR (see Figure 2). Within another framework, the assessment of the trade-offs mentioned above could well lead to a different decision.

4.2 Inlining Mechanism

The input to the inlining mechanism is the intermediate representation of a method—subsequently called the *root method*—possibly augmented with profiling information. The output is an intermediate representation of the root method in which selected `INVOKE` instructions have been replaced by inlined code (which in turn may include inlined code of nested callees).

4.2.1 The Inlining Tree

A central data structure used internally by the inlining mechanism is the *inlining tree*. The root node of this tree represents the root method. Each child node represents a call in the root method that will be inlined. These *inline nodes* can in turn have child nodes for nested inlining.

The inlining mechanism is largely divided in two passes:

1. building and decorating the inlining tree
2. traversing the inlining tree and writing the resulting IR

4.2.2 Building the Inlining Tree

Building the inlining tree is a recursive process: The compiler starts to iterate over the intermediate representation of the root method. For each `INVOKE` instruction, the compiler checks if the call could be inlined. Necessary conditions therefore are:

1. The method reference of the instruction has been resolved.
2. The result of the resolution is a monomorphic or currently-monomorphic call site.

The call site is known to be monomorphic, if any of the following is true:

- the method is declared `static`,
- the method is declared `final`,
- the method is declared `private`,
- the method is invoked via `INVOKESPECIAL`.

If any of these conditions is true, the call site is guaranteed to have only one receiver for all future invocations. Otherwise, the compiler tries to determine if the call site is *currently-monomorphic*, i.e. there is exactly one implementation of the method at the current time.

A currently-monomorphic method can become polymorphic if dynamic class loading adds a class defining another implementation of the method.

4.2.3 Determining Currently-monomorphic Methods

CACAO uses a simple conservative algorithm to determine whether a method is currently-monomorphic: Every loaded method has two flags:

IMPLEMENTED signals whether the method currently has an implementation,

MONOMORPHIC signals whether the method is currently monomorphic.

When a method is loaded, the `MONOMORPHIC` flag is initialized to `true`. The `IMPLEMENTED` flag is initialized to `false` for abstract methods, and to `true` for other methods.

In addition, each method has a pointer to the method it overwrites, which is initialized to `NULL` on loading.

When the linker determines that method S directly overwrites method G, Algorithm 1 is used to update the flags of the (in)directly overwritten methods. Thus, a method can make the state transitions:

$$\text{not implemented} \rightarrow \text{monomorphic} \rightarrow \text{multiple implementations}$$

The linker maintains the invariant that if a method S (indirectly) overwrites a method M, then:

$$\text{state}(G) \geq \text{state}(S)$$

where states are ordered as follows:

$$\text{not implemented} < \text{monomorphic} < \text{multiple implementations}$$

Algorithm 1 Algorithm for updating the `MONOMORPHIC` and `IMPLEMENTED` flags

Input:
G: the overwritten (more general) method
S: the overwriting (more specific) method

Algorithm:
S`->overwrites` $\leftarrow G$
if S is `IMPLEMENTED` and $S \neq$ `<init>`:
  while $G \neq$ `NULL`:
    if G is `IMPLEMENTED`:
      if G is not `MONOMORPHIC`:
        terminate the algorithm
      end if
      clear `MONOMORPHIC` flag of G
      break-assumption(monomorphic(G))
    else
      set `IMPLEMENTED` flag of G
    end if
    $S \leftarrow G$
    $G \leftarrow G$`->overwrites`
  end while
end if

4.2.4 Inlining Decisions

When the compiler has determined whether a call site can be inlined, it must decide whether the call site *should* be inlined. In order to make these inlining decisions, three heuristic algorithms have been implemented in CACAO:

- a depth-first algorithm for aggressive inlining,
- a breadth-first algorithm for aggressive inlining,
- a variation of the greedy heuristics used for approximately solving the KNAPSACK problem.

Aggressive Depth-first Inlining

When the depth-first algorithm finds a call site that can be inlined, it inserts the corresponding node into the inlining tree, parses the callee, and enters a recursive analysis of the callee's code. Only two conditions limit the building of the inline tree:

- Call sites below a certain depth from the root node are not inlined. For the experimental results given in Chapter 8 a maximum inlining depth of 3 was used.
- When code expansion reaches a certain limit, inlining is stopped. For the results in Chapter 8, inlining was stopped when the resulting code would have had more than ten times as many basic blocks as the original root method. The final inlining tree used was the tree before adding the callee that crossed this threshold.

A variation of this scheme was also implemented, in which inlining is completely cancelled for a root method if the code expansion threshold is reached. So the root method is either left unchanged, or *all* monomorphic call sites down to the maximum inlining depth are inline expanded. As can be seen in Chapter 8, this variation caused much less overall code expansion than the one described above, with comparable execution times in some cases. However, there were also benchmarks for which this all-or-nothing variation performed significantly worse.

Aggressive Breadth-first Inlining

The breadth-first algorithm first inlines all possible call sites in the root method. After that it inlines the call sites in the previously inlined callees. The algorithm iterates, inlining call sites at a certain depth only after all candidate call sites at lower depths have been inlined. Inlining stops when code expansion—either measured as the increase in intermediate instructions or the increase in basic blocks—reaches a threshold.

Knapsack Heuristics

In contrast to the aggressive heuristics, another algorithm was implemented that tries to select only inlining candidates with an attractive ratio of expected benefit to expected cost of inlining. The algorithm is a variant of the greedy heuristics used to approximately solve the KNAPSACK problem: The algorithm starts with a certain inlining *budget* and at each step selects the call site for inlining that has the highest ratio of benefit to cost of all candidates fitting within the budget. The costs of the selected site are subtracted from the budget. Then the selected callee is parsed and all of the contained call sites that could be inlined are added to the set of candidate call sites. The algorithm iterates until there is no candidate left that fits within the remaining budget.

The hardest problem when implementing this algorithm is calculating good estimates of the benefits and costs of inlining a call site.

4.2.5 Estimating Costs

CACAO assumes that the costs of inlining are proportional to the size of the callee's bytecode minus a small constant:

$$c = \max(N_{\text{bytes}} - c_0, 0) \tag{4.1}$$

The constant c_0 models the code size reduction by eliminating the method call. In practice this means that callees below a certain size will always be inlined. The *max* function clips costs at zero to avoid inflating the inlining budget by subtracting negative costs. (While negative costs would model the elimination of the call more accurately, experiments yielded better results with clipping at zero than without.)

More complex estimates were tried—for example by taking the number of local variables of the callee into account. However, none of the tested variations improved results over 4.1.

4.2.6 Estimating Benefits

The benefits of inlining a call site can be assumed to be roughly proportional to the number of future executions of this site. This number, of course, has to be extrapolated from data collected in the past.

As Arnold, Fink, Sarkar, and Sweeney demonstrated [4], a dynamic call graph with edge counters can provide good guiding for inlining decisions. Currently, however, CACAO does not collect such detailed profile data. One possibility would be to use the countdown field of each method to get a coarse estimate of the node counter. (For methods without loops the number would be exact.) However, for reasons discussed in the following section, method counters turned out to be quite useless for guiding inlining decisions in the current framework. Thus the final algorithm used to obtain the results in Chapter 8 assumes all call sites to have the same execution frequency.

4.2.7 Phase Changes

During development of the knapsack heuristics, it turned out that using method counters to estimate execution frequency yielded worse results than using no profile information at all. The reason for this can be found in program phase changes and the way they influence the adaptive optimization system.

For example assume a method M that contains first a loop of ten thousand iterations initializing a data structure and then a loop of one million iterations working on this data structure. M will be recompiled with optimization under the following conditions:

- The method M is active while a countdown-trap is activated.

- M itself or its callee are determined to be hot methods.

The initialization loop in M is very likely to trigger recompilation. At this point the following work loop has not even run once, so the methods called in the work loop may well have their counters at zero. Thus inlining would favor the initialization loop over the work loop, although the latter will subsequently execute one hundred times as many iterations.

4.2.8 Inlining Budget

An important parameter of the knapsack heuristics is the initial inlining budget for the root method. The CACAO implementation uses a constant inlining budget. The value was balanced for the best average effectiveness of inlining over the range of tested programs. However, no budget value yielded optimal results for all programs. This indicates that an adaptively chosen budget could further improve the effectiveness of the algorithm. Experiments were also conducted with an inlining budget proportional to the original size of the root method, but results were inferior to those obtained with a constant budget. As a future improvement, the parameters of the algorithm could be automatically tuned off-line (as demonstrated by Cavazos and O'Boyle [8]) to eliminate guesswork.

For the results shown in Chapter 8 the budget was set to 100 bytes of bytecode size. Combined with a c_0 (see equation 4.1) of 16 bytes this yielded good results on average.

Further experiments were conducted with additional limits on inlining, for example by stopping inline expansion at a certain maximum depth or code expansion factor. None of these additional constraints improved the results.

4.2.9 Inlining Prolog

Before the intermediate code of the inlined callee is copied into the IR of the caller, the inliner emits a special piece of code called the *inlining prolog*. The inlining prolog has several functions:

1. Performing a null pointer check for the `this` pointer of instance methods.
2. Copying the arguments to the callee's variables.
3. Saving the `this` pointer of the callee (for some synchronized instance methods—see Section 4.2.17).
4. Entering a monitor (for synchronized methods).

The inlining prolog also contains the special `INLINE_START` instruction at the start and the `INLINE_BODY` instruction at the end. These instructions serve as markers for later compiler passes.

4.2.10 Inlining Epilog

After the intermediate code of the callee, the inliner emits the *inlining epilog*. This code performs a `MONITOREXIT` for synchronized methods. It also contains the special `INLINE_END` instruction that serves as a marker for later compiler passes.

4.2.11 Code Rewriting

Inserting the intermediate code of the callee into the caller mostly means just copying the instruction structures. For dynamically allocated arrays referenced by the instructions, copies have to be placed in newly allocated memory. The only data that cannot be cloned in this straight-forward way are the references to variables and branch targets in the intermediate instructions. For variables, the inliner uses a variable translation map, briefly called the *varmap*. This array is indexed by the variables' indices used in the unmodified callee and contains the indices to replace them with in the resulting code.

References to branch targets must be mapped to locations in rewritten code. This is achieved by mapping references to basic blocks in the original intermediate code to the corresponding basic blocks in rewritten code. For these purposes the inliner maintains a table mapping from "old" basic blocks to "new" basic blocks. Whenever the rewriting of a block starts, the block is entered in this table. For each branch target the inliner tries to find its translation in the table. If the destination block has not yet been translated, the reference is recorded in a list of unresolved block references. When the rewriting of an inlining node is complete, the inliner resolves all block references and removes the entries from the list.

A special block reference entry is used for `RETURN` instructions in the inlined callee: The `RETURN` instruction in inlined bodies is translated to a `GOTO` to the inlining epilog. Since the epilog block has not yet been generated at that time, a special *return reference* is inserted in the block reference list. These return references are resolved when the epilog block is created.

Local Variable Coloring

When using inlining in conjunction with the simple register allocator (see Section 4.2.19), it is important to keep the total number of local variables low. The **simplereg** allocator considers all locals to be live over the whole method body and allocates them after the temporary registers. This leads

to most local variables ending up on the stack, safe for a small number that can be kept in callee-saved registers.

In order to keep the number of local variables low, the CACAO inliner translates local variable indices as follows: Every inline node is assigned a *local variable offset*. The offset for the root method is zero. When the inline tree is traversed, the offset of each node is set to the offset of its parent node plus the number of local variables used in the parent's code. In this way, inlined code sections that are not nested can use the same variable indices. When the compiler back-end is upgraded to linear scan register allocation, reusing variable indices will no longer be necessary.

4.2.12 Avoiding Basic Block Boundaries

The scope of some optimizations is limited to one basic block at a time. Also, local register allocators like CACAO's **simplereg** algorithm only perform well if little data is passed across basic block boundaries. Consequently the inlining mechanism should try to avoid introducing additional basic block boundaries where possible.

A basic block boundary at the start of an inlined callee (after the inlining prolog) is needed, if any of the following conditions is true:

- The callee contains a branch to its entry point.
- The entry point marks the start of an exception handler range, or is the start of an exception handler.
- The callee is synchronized. In this case we need to generate an exception handler covering the whole inlined body, so this is related to the second point.

A basic block boundary is needed at the end of an inlined callee (before the inlining epilog), if any of the following conditions is true:

- The callee has multiple exits.
- The callee has an exit that is not the last instruction of the method body.
- The end of the callee marks the end of an exception handler range. This is always the case for synchronized callees (see above).

The CACAO inliner separately checks these conditions for the start and the end of the inlined callee and merges the callee with the inlining prolog or inlining epilog, respectively, if possible.

4.2.13 Cross-class and Cross-package Inlining

When inlining a method of one class in a method of another class, the VM must take some particular issues into consideration. The semantics of some JVM instructions [27] are defined with respect to the *current class*, i.e. the class defining the method being executed. Examples of semantics involving the current class are:

- access checks are based on the notion of the *current class*
- calls to `protected` methods require a special check involving the current class
- the resolving algorithm specified for the `INVOKESPECIAL` instruction depends on the current class

CACAO solves these issues as follows:

- Methods, fields and class references that can be resolved lazily (i.e. without requiring any further class loading) are resolved and access checked before inlining, so the current method and class are obvious.
- For each reference that cannot be resolved lazily during compilation, CACAO creates a special data structure that records all data necessary for resolving the reference later, including the method (and thereby class) that originally contained the reference. This information is not touched by inlining, and so the correct *current class* can be used when the reference is actually resolved.

4.2.14 Exception Handling

The inlining mechanism must guarantee that the behavior of programs that throw exceptions is not modified by method inlining. In particular the following properties of JVM exception handling must remain valid:

1. When an exception is thrown, the VM first searches all exception handlers of the current method that cover the current PC in the specified order. If a matching handler is found, execution continues with the handler. Otherwise the current method activation is unwound, and the search repeats with the exception handlers that cover the call site in the calling method.
2. When an unhandled exception causes the unwinding of a synchronized method, the corresponding monitor is released.

Internally exception handlers are recorded in *exception tables*. Each *exception entry* holds the start address and the end address of the handled code range, the address of the handler, and the type of exception objects that the handler catches. When an exception is thrown, the run-time system gets the exception table for the compilation unit in which the exception occurred. The exception entries are then searched in order for the first matching entry.

It turns out that the first property mentioned above is quite easy to achieve by taking advantage of the ordered processing of exception entries: During inlining the exception tables of the calling method and the inlined methods must be combined to an exception table for the whole compilation unit. We insert the individual tables into the resulting table in topological order with respect to the inlining tree. This means that any exception entry of an inline node N occurs earlier in the resulting table than any entry of the calling node M. The order of entries within individual tables remains unchanged.

Listing 1 shows some example source code for multi-level inlining. The corresponding inlining tree is displayed in Figure 3, assuming that all calls except the one to `foo` are inlined. Table 1 shows the order of exception entries for the resulting method `one`.

Listing 1 Example methods

```
1   void one() {
2       two();
3       dos();
4   }
5
6   synchronized void two() {
7       three();
8       tres();
9       three();
10  }
11
12  void dos() {
13  }
14
15  void three() {
16      foo();
17  }
18
19  void tres() {
20  }
```

Listing 2 Example inlining result

```
void one()
{
    synchronized { // two
        { // three-1
            foo();
        }
        { // tres
        }
        { // three-2
            foo();
        }
    }
    { // dos
    }
}
```

handlers of "three-1"
handlers of "tres"
handlers of "three-2"
handlers of "two"
`MONITOREXIT`-handler for "two"
handlers of "dos"
handlers of "one"

↓ direction of processing

Table 1: Exception handler entries for method "one" after inlining

4.2.15 Building Stack Traces

In order to build stack traces with line number information, the VM must maintain a mapping from positions in compiled code to the source line numbers. As method inlining has to be a transparent optimization, the stack traces created by a program must be invariant with respect to method inlining.

CACAO takes the following basic approach to mapping line numbers: For each compilation unit the compiler creates a *line number table* containing two columns: the first column holds the machine code position, the second column holds the corresponding line number. Whenever the current source line number changes during code generation, an entry is added to the table, recording the current machine code position and the new source line number. The resulting table is sorted by machine code position and each entry (p_i, l_i) means that the machine code positions from p_i up to, but not including, p_{i+1} (or the end of the compilation unit for the last entry) have source line number l_i.

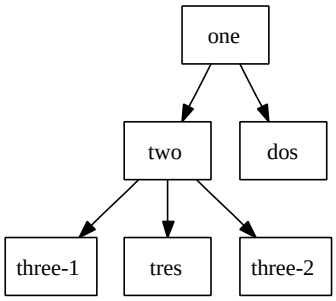

Figure 3: Example inlining tree for the compilation of "one"

Inlining requires some extensions of the line number table. When a stack trace is built for a position in an inlined method body, the virtual machine must be able to determine the name of the inlined method and the position of the (eliminated) call site. For this purpose the code generator inserts special markers into the line number table whenever it encounters the start or the end of an inlined method body.

Figure 4 shows the structure of the table for an inlined call in source line x. The body of the inlined method has source lines m to n. The start marker is identified by the special line number -2 and points to the start of the inlined code. The end marker (identified by the special line number -1) encodes the source line of the inlined call, a reference to the method that has been inlined, and the address of the *start* of the inlined code.

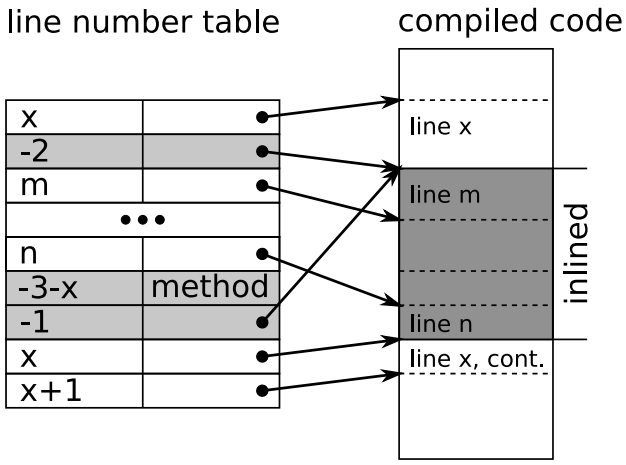

Figure 4: Line number table with inlining markers

When a stack trace needs to be built, CACAO iterates over the stack frames and searches the line number table of each frame for a line containing the current PC. The search moves backwards through the line number table

and selects the first entry for which $p_i \leq PC$.

An inlined method body first appears to the backward search algorithm like a negatively numbered line covering the code of the inlined body. If the PC is determined to be inside this code range, the algorithm recurses to find responsible source lines inside the inlined body. After the recursion returns—and thus all entries for lines inside the callee have been added to the stack trace—an entry with the source line of the inlined call is added to the stack trace and the algorithm terminates.

4.2.16 Null Pointer Checks

The semantics of the `INVOKEVIRTUAL` and `INVOKEINTERFACE` instructions demand that the instance argument is checked, and a `NullPointerException` is thrown if the instance is `null`.

CACAO implements this check efficiently by relying on the MMU (memory management unit) hardware: In order to perform dynamic dispatch, `INVOKEVIRTUAL` and `INVOKEINTERFACE` instructions need to load the vftable pointer from the object instance. Since the vftable pointer resides in the object header, an attempt to read the vftable pointer from a `null` instance is guaranteed to trigger a segmentation fault. The signal handler of the thread then constructs and throws the required `NullPointerException`.

For inlined calls the dynamic dispatch has been eliminated. Thus, in order to maintain correct semantics, we must perform an explicit null pointer check for inlined calls to instance methods.

`INVOKESPECIAL` instructions always need the explicit null pointer check, whether the instruction is inlined or not. The compiler can, however, eliminate the null pointer check for calls to `<init>` methods, as the static constraints on Java bytecode guarantee that instance initializers are only called for references to uninitialized objects, which can never be `null`.

4.2.17 Synchronized Methods

The CACAO inliner handles synchronized methods in a similar way as Java compilers deal with the

```
synchronized (obj) {
    ...
}
```

construct: A special exception handler (with catch type *any*) is created to protect the inlined method body, including any exceptions handlers of the inlined method. When invoked, this exception handler performs the following steps:

1. Save the exception object in a local variable.

2. For instance methods, fetch the `this` pointer of the inlined method. For static methods, take the `java.lang.Class` object of the method's class.

3. Perform a `MONITOREXIT` operation on the retrieved reference.

4. Throw the saved exception object.

The `this` pointer of instance methods can be read from the (bytecode) local variable 0, provided the value of this variable is not modified in the callee. In the more general (but uncommon) case of writes to local variable 0, a separate local variable must be created for synchronization which receives a copy of the `this` pointer in the inlining prolog.

4.2.18 Argument Handling

When a method is inlined, the inlining prolog must copy the actual arguments of the call to the variables representing the formal arguments in the callee. In order to make this more efficient, the inliner could analyze the use of the formal arguments in the callee and, for example, directly replace read-only arguments with the actual argument variables. A similar optimization could be done for the return value of an inlined method. However, these optimizations can be viewed as limited special cases of copy propagation. In the spirit of separation of concerns, the CACAO inliner just inserts straight-forward `MOVE` instructions in the prolog and epilog code and leaves the optimization of the argument passing to a subsequent copy elimination pass (see Section 3.2).

4.2.19 Effects on Local Register Allocation

One of the register allocators available in CACAO is a simple local allocator, called **simplereg**, that works in three steps on a method [25, 24]:

1. Allocate the *interface variables*. These are the variables holding the contents of the Java stack at basic block boundaries.

2. For each block, allocate the block-local temporary variables, assuming interference with all interface variables used or defined in this block.

3. Allocate the local variables of the method, assuming interference with all interface variables and block-local variables.

Characteristics of this algorithm are:

- Introducing an interface variable greatly increases register pressure on block-local variables, as each block-local variable is assumed to interfere with each interface variable.

- Local variables are very likely to be spilled, since they are allocated in the final step.

Since inlining tends to increase the number of values that are live at block boundaries and the number of local variables per compilation unit, we may expect a degradation of register allocation when using **simplereg** on the results of the inline transformation.

For an empirical assessment, the numbers of variables allocated to registers where compared for the **javac** benchmark of the SPECjvm98 suite. In code compiled by the baseline compiler, about 73% of all variables were allocated to registers. In code recompiled by the optimizing compiler it was only 63%. These numbers confirm that **simplereg** is not well suited for the optimizing back-end. Better results may be expected with the linear scan register allocator, which is currently being developed.

4.3 Recording Assumptions

When speculative inlining is performed, the compiler records the assumptions made in the method database. Thus assumptions can be tracked and invalidated when they become false because of new classes being dynamically loaded.

For every loaded method the method database keeps a list of assumptions. When a currently-monomorphic call to the method m is inlined, the compiler records the assumption *monomorphic*() in m's assumption list. The empty parentheses illustrate that this assumption has no further parameters. The system could be easily extended to assumptions depending on parameters. For example if a call to method m is determined to be currently-monomorphic for a statically inferred receiver type T, the compiler would record the assumption *monomorphic*(T).

For each assumption the following data is kept:

- the kind of assumption, e.g. *monomorphic*,
- parameters of the assumption (currently none),
- the context of the assumption, i.e. the root method of the compilation unit which depends on the assumption.

When the linker overwrites a method such that multiple implementations become defined, it checks if any recorded *monomorphic* assumptions are violated. Methods that depend on broken assumptions are put on a work list. When linking is done, and before the new class can be used by Java code, the code of all methods on the work list is invalidated.

Chapter 5

Local Subroutine Inlining

5.1 Introduction

In order to allow code reuse within Java methods, and thereby decrease bytecode size, Java supports the notion of *local subroutines*. Local subroutines are defined by special semantics of the `JSR`, `ASTORE`, and `RET` instructions [27]:

1. The `JSR` D instruction pushes the address of the following instruction onto the operand stack and then jumps to the given destination D. The stack slot created by `JSR` has the special primitive type *returnAddress*.

2. The `ASTORE` L_i instruction can be used to pop a *returnAddress* value off the stack and store it in the local variable L_i.

3. Finally, the `RET` L_i instruction returns to the *returnAddress* contained in the local variable L_i.

The code executed after the `JSR` but before the return to the instruction following the `JSR` is called a *local subroutine*. Thus the notion of a local subroutine is defined dynamically. While there are certain static constraints on `JSR` and `RET` instructions, there are no statically well-defined boundaries between the code making up a local subroutine and "surrounding" code. Also the `JSR` and `RET` instructions are not required to be properly nested.

An important feature of local subroutines that is explicitly required by the JVM specification [27] is that a local subroutine be polymorphic with respect to the types of local variables not used in the subroutine. As a consequence, there may be points in a method where the basic type[1] of a local variable cannot be determined statically.

[1] Here, "basic type" refers to the distinction between integer, float, double, reference, and *returnAddress* types.

5.2 Implications of Local Subroutines

Local subroutines have implications for many parts of the virtual machine. The following sections list some areas in which complexity is increased by the presence of local subroutines.

5.2.1 Control-flow Graph

As the target of a `RET` instruction is determined dynamically, the control-flow graph must contain edges from the `RET` to every possible target. Multiple `JSR` instructions to the same local subroutine introduce control-flow merges. Thus local subroutines reduce the precision of the control-flow graph, as the graph will often represent many paths that can never be taken during program execution.

5.2.2 Verification of Local Subroutines

Local subroutines significantly complicate bytecode verification because of their effect on the control-flow graph and the requirement that types of unused local variables may not be merged at entry to the local subroutine.

Coglio [12] developed an elegant technique for verification of local subroutines that solves these problems. CACAO performs local subroutine elimination based on Coglio's technique, as will be described in section 5.3.1.

5.2.3 Effects on On-Stack Replacement

The semantics of the `JSR` and `RET` instructions interfere with on-stack replacement. Before jumping to its target, the `JSR` instruction pushes the address of the following instruction onto the operand stack. This *returnAddress* value is later used as the target of a `RET` instruction. If on-stack replacement of the method occurs after the `JSR` but before the `RET` instruction, the *returnAddress* value will refer to a position in invalidated code and the `RET` instruction will subsequently load this invalid program counter.

Several approaches could be taken to solve this problem:

- The `JSR` instruction could be changed to store an integer label identifying the following instruction, and `RET` would use a jump table to calculate the target address from this integer.
- On-stack replacement could translate live *returnAddress* values to the corresponding addresses in replacement code.
- On-stack replacement could keep invalidated code in memory and patch all possible targets of `RET` instructions with jumps to the corresponding positions in replacement code.

Obviously these are all rather complex workarounds and it would be preferable to eliminate local subroutines in order to avoid these problems altogether.

5.3 Elimination of Local Subroutines

Local subroutine elimination is not performed as an optimization in its own right, but for its large down-stream benefits. A significant source of complexity is removed when subsequent compiler passes do not have to deal with the intricacies of local subroutines. The following section describes the technique used for eliminating local subroutines in CACAO.

5.3.1 Type-based Specialization Approach

One effective way of handling local subroutines [12] is to treat each return address i that is created by a `JSR` to address j as an instance of a special type $R_{i,j}$. The compiler can then perform abstract interpretation of the bytecode (or of the intermediate representation, as is the case in CACAO) using special merging rules for the family of *returnAddress* types:

$R_{i_1,j_1} \sqcup R_{i_2,j_2} = \bot$	for $j_1 \neq j_2$
$R_{i_1,j_1} \sqcup R_{i_2,j_2} =$ not merged	for $j_1 = j_2 \wedge i_1 \neq i_2$
$R_{i_1,j_1} \sqcup T = \bot$	for any non-*returnAddress* type T

The first row enforces the static guarantee on Java bytecode that the control-flow of two distinct local subroutines j_1 and j_2 may not be merged to a single `RET` instruction.

The last row derives from the fact that *returnAddress* types are assignment-incompatible with all other JVM types.

The middle row is the key to local subroutine elimination: When a stack slot or a local variable contains different—but compatible— *returnAddress* values along the control-flow edges reaching a basic block, then we do not merge these types. Instead we duplicate the basic block and create specialized versions for each combination of *returnAddress* types that stack slots and local variables may have at the entry to this block.

This way, when we reach a `RET` L_k instruction, and the local variable L_k has type $R_{i,j}$ at this point, we know that the `RET` instruction will return to address i, and we can thus replace `RET` L_k with `GOTO` i. (If L_k is of any other type, a VerifyError is thrown.)

5.3.2 Code Expansion

One problem with the type-based specialization approach is that it can lead to exponential code expansion in some cases. These cases do not occur in

code created by Java compilers but they are easy to construct manually. The example code in Listing 3 causes the creation of over one million (4^{10}) basic blocks.

This code expansion problem is not restricted to the type-based specialization approach, but it occurs with any algorithm that eliminates `JSR` and `RET` by replacing them with `GOTO` instructions.

Two approaches for dealing with such cases would be:

- Retaining local subroutines and falling back to the baseline compiler in cases where code expansion reaches a certain threshold.
- Transforming local subroutines to compiler-generated methods. This approach would require significant changes to the virtual machine infrastructure, however, as the generated methods would have to be completely transparent to the executing program.

Listing 3 Bytecode causing exponential code expansion with `JSR` elimination

```
1           jsr s1
2           jsr s1
3           jsr s1
4           jsr s1
5           return
6
7   s1:
8           astore 1
9           jsr s2
10          jsr s2
11          jsr s2
12          jsr s2
13          ret 1
14
15  s2:
16          astore 2
17          jsr s3
18          jsr s3
19          jsr s3
20          jsr s3
21          ret 2
22
23  ; ...
24
25  s10:
26          astore 10
27          iinc 0 1
28          ret 10
29
```

Chapter 6

Code Replacement

6.1 Requirements of Adaptive Optimization

The requirements on the replacement mechanism when used for adaptive optimizations can be summarized as follows:

installing code Newly compiled versions of methods must be installed to replace older versions. As the new versions are usually better optimized, the replacement should happen as soon as possible. An important case are methods that are rarely entered or left but contain frequently executed loops. Such methods should be replaced *on-stack*, i.e. while they are activated.

invalidating code When code optimized on preliminary assumptions becomes invalid, this code *must* be replaced before it can create an inconsistent program state. This requires *replacement traps* that can be set at the start of invalidated code regions.

re-allocation of values Old and new versions of code may differ in the allocations of variables. For example, one version may keep a value in a register, while another version puts the same value on the stack.

re-grouping of stack frames By changing inlining decisions, the grouping of source-level frames into machine-level stack frames can change. The replacement mechanism must be able to translate in both directions between code that performs a machine-level call, and code that is inlined within the same compilation unit.

6.2 Requirements of Exact Garbage Collection

Exact garbage collection has several requirements that overlap with features of the replacement mechanism:

- Program execution must be suspended at certain *safe points* where all live references can be exactly determined.
- The garbage collector must be able to set *GC traps* at safe points in other threads in order to trigger their suspension.
- If the garbage collector performs compaction, it needs a way for redirecting references to the new locations of objects.

These requirements, however, are not a subset of the requirements of replacement for adaptive optimization. There are several areas where exact garbage collection increases the demands on the replacement mechanism:

GC traps To ensure timely garbage collection, GC traps must be installable at all points where execution can leave a method. This includes all invocations and the method exit.

source state The garbage collector needs the full set of live references. This means that *all* frames on the stack must be examined. Thus the recovering of the source state may not stop at intermediate stack frames of native code, as is the case for replacement.

redirecting references While replacement after recompilation is concerned with changing the *allocation* of values, a compacting GC changes the *values* of references. The new values need to be written back to registers and stack frames. This update must also be performed for the *full* stack of each thread, so stack frames beneath invocation of native code must also be processed.

6.3 Replacement Strategies

As soon as we make use of recompilation, replacing old code with new versions becomes an issue. We can distinguish two basic kinds of demands requiring code replacement:

- A piece of code was recompiled because it has become invalid. In this case we *must* ensure timely replacement of the old code.
- A method has been recompiled to a more optimized version, though the old code is still valid. In this case we *should* replace the old code as soon as possible in order to get better performance.

Independently we can classify the replacement strategy on how we want to deal with existing references to old code:

- *Eager replacement*: We replace all references to the old code at once, no matter whether, or when, these references would be used in the future.
- *Lazy replacement*: References to old code may remain in existence indefinitely, but we take measures that stale references are updated in time when they are used.

This distinction is important if we plan to release memory used by old code. In this case we can only use eager replacement strategies.

6.4 Replacement of Future Invocations

When a method is recompiled, all future invocations of this method should use the newly compiled version of its code. In fact, if the method is recompiled because the current version became invalid, future invocations *must* use the new code, so that the program's state does not become inconsistent. The actions necessary to redirect future invocations depend on whether calls to the method are dispatched dynamically or statically. Note that this is not a property of the method itself, but of each individual call site, as some calls to a virtual method may be bound statically, while others are not.

6.4.1 Eager Replacement of Dynamically Dispatched Methods

In CACAO virtual calls are dispatched using virtual method tables. Each class has a virtual method table containing the entry points of all methods implemented by the class, or inherited from superclasses. In addition each class can have a number of *interface method tables*. Every interface is assigned a unique integer index, starting with zero. For each class that implements one or more interfaces, the maximum index of these interfaces is determined. The class gets an *interface table* that can be indexed up to this maximum value. For each implemented interface, this table holds a pointer to an *interface method table*. The interface method table contains the entry points of the methods implementing the interface. Note that these are always methods that also occur in the virtual method table of the class.

When the code of a virtual method is replaced, the VM must update the entry point stored in the virtual method tables of the defining class and of all classes that inherit the method, and in all interface method tables of those classes that refer to this method.

There are two basic approaches for updating the entry points:

1. Traverse the class hierarchy of the defining class and its subclasses moving from less derived to more derived classes. For each class,

search the method tables for entries referring to the old entry point and update them. The traversal can be terminated at each class that overwrites the method being replaced.

2. For each method, keep a list of the method table entries referring to this method. When the method is replaced, iterate over this list and update the table entries.

We are facing a speed/memory trade-off between these approaches. The second approach avoids checking any table entries that are not related to the method being replaced. On the other hand it requires additional memory for each loaded method.

An important problem with both approaches is synchronization. We must take care that neither concurrent changes to the class hierarchy, nor concurrent replacements of methods can interfere.

6.4.2 Eager Replacement of Statically Bound Methods

The entry points of statically bound methods appear as constants in the compiled code of their callers. Thus we cannot redirect all future calls by changing any single value. The following approaches come to mind:

- Whenever the compiler binds a method statically, it could record the position of the call site in a list associated with the method. To redirect future calls to the method we could then iterate over this list and update all existing call sites to the new entry point.

- We could install a jump to the new entry point at the start of the old code. This way calls to the old entry point would be redirected to the new code. However, the cost of the additional jump would have to be payed each time the call site is used.

6.4.3 Lazy Replacement of Future Invocations

Instead of updating all references as soon as new code becomes available, we can implement a *lazy* replacement strategy: A trap is set at the start of the old code that triggers the following actions:

1. The return address is read in order to find the position of the calling code.

2. In the case of an instance method, the instance argument is determined.

3. The call site is inspected, and depending on the type of call the data used for dispatch is modified:

- For statically bound calls, the destination address of the call is updated.
- For a virtual class method call, the method's entry in the virtual method table is updated.
- For an interface method call, the method's entry in the interface method table is updated.

This is very similar to the mechanism that CACAO already uses to implement lazy compilation. The *compiler stubs* that are initially installed to trigger compilation are replaced with the compiled code by patching statically bound calls and updating method table entries in the way described above.

6.5 On-Stack Replacement

In the context of on-stack replacement it is fundamental to distinguish two different representations of program state:

1. the **machine-level** state, hereafter called the *execution state*, which depends on compiler optimizations,
2. the **source-level** state, hereafter called the *source state*—a representation that is independent of any optimizations used. The source-level state is the state that would have been created by interpreting the unmodified bytecode of the program up to the current point. Thus all optimizations that preserve the semantics of the program are transparent with respect to the source-level state. This allows the source-level state to serve as a common ground for translating state between differently optimized versions of the same program.

The following sections give detailed definitions of these concepts.

6.5.1 Execution State

We define the *execution* state to be the machine-level snapshot of a thread at a certain time. The execution state of a thread comprises:

- the current values of the CPU registers,
- the contents of the machine stack of this thread.

6.5.2 Source State

The source state of a thread comprises the currently active stack frames of the Java virtual machine stack. (Our notion of *source frame* is equivalent to the *JVM scope descriptor* in [19].) For each source-level stack frame the following data are given:

- the currently executing method,
- the bytecode position within this method,
- the types and values of local variables that are currently live in the frame,
- the types and values of the operand stack slots that are currently live in the frame,
- the object that the method synchronizes on, if any.

The most important property of the source state is that at any given point in the execution of the program the source state is independent of any past optimization decisions. In other words, all optimizations are transparent with regard to the source state. Thus when we revert optimization decisions at some point, program correctness is guaranteed if we replace the current execution state by a new execution state corresponding to exactly the same source state.

In order to be able to recreate an execution state from the source state, we also need the following information:

- the values of all saved registers before the activation of the bottommost stack frame of the source state,
- the value of the stack pointer at this time.

6.5.3 Partial Source States

When replacing a compiled method—as opposed to finding the GC root set—we are usually not interested in the full source state down to the first stack frame activated within the thread. For example, it is not feasible to relocate the stack frames of native methods, so it makes no sense to traverse them when deriving the source state. Thus we just derive the source state down to the lowest stack frame we want to replace, and record the values of the saved registers and the stack pointer at this point, so we are able to build the new execution state from this point upwards, while leaving the machine stack below unchanged.

6.5.4 Native Frames

Typically the outermost Java method has been invoked by native code. However, there are also invocations of native code interspersed with invocations of Java code on the stack. We call the activation records of native code *native frames*. Native frames provide natural boundaries for on-stack replacement. We do not know the layout of native frames, so we cannot relocate them. (Even if we knew the layout, there could still be references to variables in the native frame from other native code, so relocation would still be impossible.)

However, to fulfill the requirements of the exact garbage collector, the replacement mechanism must provide support for traversing through native frames. For all stack frames below the innermost native frame, only restricted functionality is provided:

- The layout[1] of stack frames may not be changed. This also precludes mapping to other versions of code.
- Only reference values[2] may be read out and written back.

When a native frame is traversed while recovering the source state, saved registers pose a problem: Method activations above the native frame may keep values in saved registers. The native code is responsible for either preserving or saving these registers, but we do not know whether or where the values have been saved on the stack. However, the exact garbage collector requires that:

- all live reference values can be determined,
- live references can be redirected during compaction.

To fulfill the first requirement, we must be able to find the values of saved registers. Thus before any invocation of native code, all general-purpose saved registers must be copied to a structure of known layout. The second requirement demands that when execution returns from native code, the values in this structure must be copied back to the saved registers. (The native code itself guarantees that saved registers are restored to their value at the time of invocation. However, because of GC compaction, these might not be the appropriate values any more.)

In order to build correct stack traces, CACAO already does special bookkeeping for calls into native code: Whenever control is transferred from

[1] Theoretically we must only require the size of the stack frames between native frames to remain unchanged. Practically this precludes any changes to the stack frame layout.

[2] Technically the restriction is: only values that can be kept in general-purpose registers. See the following discussion of saved registers.

JIT-compiled code to native code, a piece of generated wrapper code creates a `stackframeinfo` structure recording the values of PC, stack pointer, and PV[3] at the invocation. For supporting the exact GC, we augment this structure with slots to store the callee-saved general-purpose registers. Callee-saved floating-point registers are not included, as the garbage collector does not need to read or change their values, and saving them would incur additional runtime costs.

6.5.5 Replacement Points

A *replacement point* is a position in compiled code where we have sufficient information to reconstruct the source state from the execution state taken at this point. We will hereafter use the term *replacement point* to refer to both the actual compiled code position and the data structure associated with this position that is needed for the transformations between execution state and source state. The VM needs to store the following data for each replacement point to make this transformation possible:

- the machine code position of this point,
- the method that contains the source position for this point,
- the bytecode index of the source position,
- the type and allocation of live local variables,
- the type and allocation of live stack slots,
- whether the code at this point is synchronized on a Java object, and where to find this monitor object (the register or stack slot position),
- if this point is within an inlined method body, a reference to the replacement point corresponding to the start of the inlined body within the calling method.
- match conditions and rules for extracting values as described in section 6.5.10, if this replacement point is not uniquely identified by the method and the bytecode index.

We call a replacement point P *mappable* if it is guaranteed that all compiled versions of the containing method will have a replacement point corresponding to the same program point as P. Replacement points used for switching between differently compiled versions of a method must be mappable. Replacement points that are only used for GC traps are not

[3]The PV register contains the *procedure value*, i.e. the entry point of the currently executing compilation unit.

required to be mappable. (For example, GC traps located at method exits have no counterpart in compilation units that inline this method.)

Figure 5 visualizes how a replacement point captures the allocation of live ranges at a call site. Notice that a replacement point at a call site has two aspects, depending on how it is reached during replacement: If execution is trapped at the point of invocation, all variables reaching the call site are live, including the arguments to the call. If, on the other hand, the replacement point is reached in the course of unwinding activation records, only the variables living through the call are guaranteed to be live. This has important consequences for the instance argument of non-static methods: When such a method is active while a replacement point is reached, there is no guarantee that there is a live variable referring to the object instance of this invocation.

Figure 6 shows the equivalent replacement points in the case that the call site has been inlined.

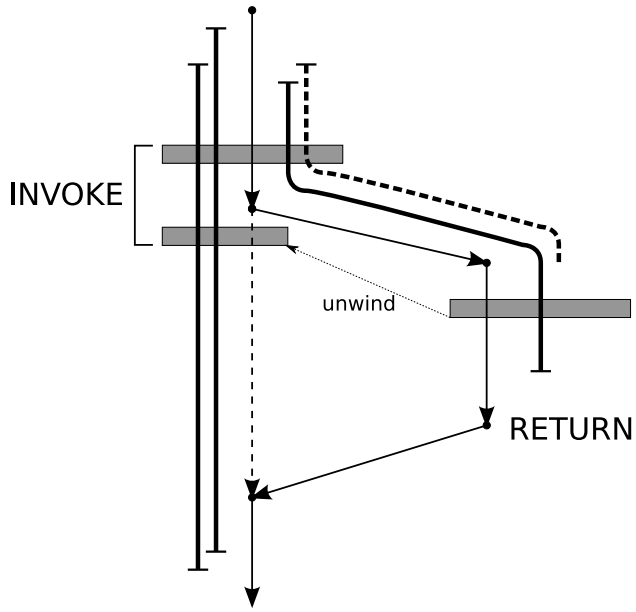

Figure 5: Replacement points and live ranges at a call site

6.5.6 Replacement Traps

Once we have replacement points available, the question arises how to trigger replacement when a certain replacement point is reached. For this purpose we introduce *replacement traps*. A *replacement trap* is a modification of the machine code at a replacement point that causes the thread to depart from normal control-flow and enter the replacement mechanism. If it is possible

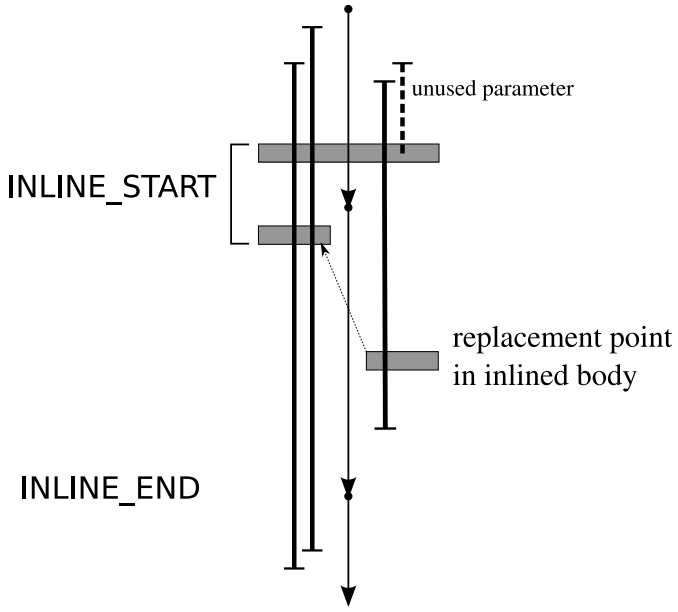

Figure 6: Replacement points and live ranges with inlining

to set a replacement trap at a replacement point, we call the point *trappable*.

There are several ways that replacement traps can be realized:

- By patching a branch instruction into the code, or
- by modifying the target of an already existing branch, or
- by patching an instruction into the code that triggers a hardware exception.

CACAO implements replacement points by writing an unconditional branch instruction into the existing code. The target of the branch instruction is the corresponding *replacement stub*. The replacement stub is a small piece of code that is prepared for each replacement point when the method is compiled. The stub calls the `replacement-out` function of the architecture-dependent layer with a reference to the replacement point that was reached as an argument. This architecture-dependent function is responsible for saving all CPU registers and filling an execution state structure. Finally it calls the architecture-independent entry point of the replacement system.

In future versions of CACAO it is planned to use hardware exceptions for triggering replacement traps. This would remove the need for replacement stubs.

6.5.7 Countdown Traps

A special kind of replacement traps are *countdown traps*, which are used for instrumenting code generated by the baseline compiler. For a countdown trap, the compiler generates code to decrement a counter variable and a conditional branch. If the counter becomes negative, the conditional branch transfers control to the replacement stub and thus activates the replacement system. Otherwise, executions proceeds normally.

Each method has an associated counter variable that is initialized to a fixed threshold. The baseline compiler sets countdown traps at the method entry and at the targets of all backward branches using this counter variable. Thus, if a method is frequently entered or contains a frequently executed loop, the replacement system will be activated for this method, which allows recompilation of the method to more optimized code.

6.5.8 Location of Replacement Points

The program locations where the compiler must create replacement points are determined by the sum of all requirements on the replacement mechanism. These requirements can be summarized in the following groups:

- timely replacement after recompilation,
- re-grouping of stack frames for inlining,
- guaranteed replacement of invalidated code,
- requirements of the exact garbage collector.

Additionally replacement points must satisfy some conditions:

- Non-GC replacement points must be mappable.
- At GC replacement points it must be possible to recover the full source state.

The notion of *timely replacement* needs to be defined more precisely. We will speak of timely replacement if the following property holds: After a compilation unit C is marked for replacement, each thread will execute each instruction in C at most once before entering the replacement mechanism. Note that a thread may execute an arbitrary number of instructions *outside* of C, or block for an unlimited amount of time, before entering the replacement mechanism, without breaking this property.

A natural choice to guarantee timely replacement is to place replacement points:

- at method entry,

- at the target of each backward branch (including exception handlers causing control-flow to move backwards)

Proof: At the time t_0 a compilation unit C is marked for replacement, C can be either active or not active in a thread T. If C is inactive, T can only execute an instruction within C if it enters C first. As there is a replacement point at the entry point, this case cannot lead to timely replacement being violated. In the case of C being active in T, let p_t denote the maximum PC of all instructions of C executed after t_0, but before t. Let t_x denote the earliest point in time at which timely replacement is validated for C, where $t_x > t_0$. In order to violate timely replacement at t_x, the program counter at that time must have a value of $PC(t_x) \leq p_{t_x}$ As no replacement point in C has been reached by the time t_x, we know that the subsequence of the $PC(t)$ lying within C with $t_0 \leq t \leq t_x$ does not contain the entry point of C and that it is monotonically rising. Thus $PC(t_x) > p_{t_x}$, which contradicts our assumptions. □

Guaranteed replacement of invalidated code requires that there is a replacement point at the start of each code section that can become invalid. (More precisely, we must require that from the time of invalidation of a code section S, no thread may enter S without reaching a replacement point in the same compilation unit first. So we assume that if a thread T is within S at the time of invalidation, T may complete the current execution of S without any harm to program correctness.)

Since the exact garbage collector needs full source states, we must demand the following: Whenever a thread T triggers a collection, or reaches a replacement trap, each activation on the stack of T has been invoked from a replacement point.

Location	general replacement	inlining	GC
back branches	yes	no	yes
calls	no	some	yes
inline start	lazy	some	yes
method entry	lazy	no	no
method exit	no	no	yes

Table 2: Locations of required replacement/GC points

Table 2 summarizes where replacement points are needed: Timely replacement generally requires replacement points at backward branches and—if lazy replacement is used—at method entries and at the start of inlined bodies (so the method entry points are mappable.) Inlining requires replacement points at speculatively inlined call sites and at the corresponding non-inlined call sites for deoptimization. The exact garbage collector requires replacement points at method exits and at all points where stack

frames are created. Replacement points at the start of inlined bodies are needed, too, in this case, as each replacement point only describes the part of stack frame used by the containing inlined section.

6.5.9 Compilation Units

In a VM supporting recompilation it is necessary to distinguish between a method as defined within a class, and the compiled code representing this method. The former is unique over the lifetime of the class, while the latter changes when the method is recompiled, and there may even be different compiled versions of one method in use at a time. While the unit of compilation usually is a method, inlining produces compilation units that also contain compiled inlined bodies of other methods.

In CACAO a compilation unit is represented by a structure called *codeinfo*. Each codeinfo contains:

- a reference to the root method,
- the machine code entry point,
- the size of the machine code of this compilation unit,
- the list of replacement points in this compilation unit.

6.5.10 Mapping Replacement Points

One requirement for obtaining the source state is that the position of any replacement point in the compiled code can be mapped to the corresponding position in the bytecode, which is part of the source state. Vice versa, when creating the execution state for the replacement code, the VM must be able to map this position in the bytecode back to a replacement point in the new compiled version. These mappings can be hard to obtain, since transformations that the compiler performs may break the one-to-one correspondence of bytecode positions to native code positions.

For example, local subroutine elimination (see Chapter 5) and loop unrolling can lead to multiple native code positions corresponding to a single bytecode position. The underlying principle of these transformations is that state which is held in local variables by the original bytecode is encoded into the native program counter. In order to correctly derive the full source state, the VM must reconstruct the values of these local variables from the value of the native program counter.

On the other hand there may be transformations mapping several bytecode positions to a single native code position.

We need to take several measures to obtain unambiguous mappings of replacement point positions:

1. The internal representation must provide a way to find the original bytecode index for every intermediate instruction that may become the site of a replacement point.

2. If multiple replacement points P_i share the same original bytecode index, the following must hold:

   - For each of the P_i there is a *match condition* C_i on the values of live stack slots and local variables, such that for each possible combination of values exactly one of the conditions C_i is true, thereby selecting the corresponding replacement point P_i.
   - For each of the P_i the VM stores the information necessary to extract values of stack slots and local variables that have been encoded into the program counter at the point P_i.

Example 1. Consider a simple case of JSR elimination: The program in Listing 4 is transformed into the program in Listing 5. This would lead to the replacement points in Table 3.

Listing 4 Replacement point in a local subroutine

```
1:      JSR 10
2:      JSR 10
3:
...
10:     ASTORE L2
11:     <replacement point> (bytecode index 11)
12:     RET L2
```

Listing 5 Replacement points after JSR elimination

```
1:      GOTO 20
2:      GOTO 30
3:
...
20:     NOP
21:     <replacement point P1> (bytecode index 11)
22:     GOTO 2
...
30:     NOP
31:     <replacement point P2> (bytecode index 11)
32:     GOTO 3
```

Example 2. Another interesting case arises with loop unrolling. The loop in Listing 6 is unrolled four times, as shown in Listing 7. This gives rise to the replacement points listed in Table 4.

Point	Bytecode Index	Condition	Values
P1	11	**if** $L_2 = \text{ret}_2$	$L_2 = \text{ret}_2$
P2	11	**if** $L_2 = \text{ret}_3$	$L_2 = \text{ret}_3$

Table 3: Replacement points after JSR elimination

Listing 6 Replacement point in a loop

```
for (L2 = 0; L2 < 4*n; ++L2) {
        <replacement point P>
        ...
}
```

6.5.11 Recovering the Source State

When a replacement point has been reached, we first take a snapshot of the current execution state. Then we can recover the source state of the thread using an iterative algorithm similar to the creation of a stack trace, or the unwinding of the stack in the course of an exception.

For visualizing the algorithm it is useful to imagine the (partially recovered) source state on top of the execution state, and between them as a dividing line the *representation frontier*. Initially the source state is empty, and the execution state is a full snapshot of the current thread. The algorithm now recovers one source frame after the other, with each step adding to the source state and shrinking the execution state by pushing the representation frontier "down" towards the bottom of the stack.

Note that while the execution state shrinks conceptually, the actual stack contents are not modified at this point.

Figure 2 shows the algorithm in detail. The functions referred to have the following semantics:

pop-native-frame(E,I) Removes the native frame described by I from

Listing 7 Replacement points in an unrolled loop

```
for (L' = 0; L' < n; ++L') {
        <replacement point P0>
        ...
        <replacement point P1>
        ...
        <replacement point P2>
        ...
        <replacement point P3>
        ...
}
```

Point	Bytecode Index	Condition	Values
P0	11	**if** $L_2 \equiv 0(\text{mod}4)$	$L_2 = 4L'$
P1	11	**if** $L_2 \equiv 1(\text{mod}4)$	$L_2 = 4L' + 1$
P2	11	**if** $L_2 \equiv 2(\text{mod}4)$	$L_2 = 4L' + 2$
P3	11	**if** $L_2 \equiv 3(\text{mod}4)$	$L_2 = 4L' + 3$

Table 4: Replacement points after loop unrolling

the execution state E and returns the remaining execution state and a native source frame.

in-java-code(E) Checks if execution state E has a PV value corresponding to a Java method.

get-code(E) Returns the compilation unit that is active in the given execution state E.

find-replacement-point(E,C) Finds the replacement point in compilation unit C that corresponds to the current program counter in execution state E.

read-source-frame(E,C,P) Recovers a source frame from execution state E. C is the current compilation unit, P is the replacement point that the program counter is at.

find-inline-start(C,P) Finds the replacement point at the start of the innermost inlined region containing the replacement point P in compilation unit C.

find-caller(E,C) Find the compilation unit that called the active compilation unit C in execution state E.

pop-activation-record(E,C) Remove the activation record of the active compilation unit C from execution state E. Returns the reduced execution state.

6.5.12 Rebuilding the Execution State

When we have successfully recovered the source state, we must now perform the dual operation of building up a new execution state representing the same source state in the context of the modified optimization decisions we are switching to. We rebuild the execution state for one source frame at a time, each step adding to the execution state and shifting the representation frontier "upwards" until all frames of the source state have been encoded into the new execution state.

Figure 3 shows the algorithm for rebuilding the execution state. The functions referred to have the following semantics:

push-native-frame(E,F) Push the native frame F onto the execution state E. This function does not actually manipulate any stack contents, as the layout of the native frame is unknown. Instead, registers in the execution state are updated, and the values of saved registers are written to the stack frame info referred by frame F.

push-activation-record(E,F,P) This function mimics the stack effects of a method call and the subsequent saving of callee-saved registers.

get-target-replacement-point(F) Returns the replacement point to which frame F has been mapped.

write-source-frame(E,F,P) Write the values given in source frame F to the registers and stack locations in execution state E using the allocation info of replacement point P.

6.5.13 Dealing with Stack Expansion

Stack frames of the original code and the recompiled code can differ in size. Thus when rebuilding the execution state, the rewritten stack can require more space than the original stack. This poses a problem for the replacement system, as it rewrites the stack of the current thread and thus would overwrite its own stack while rebuilding the execution state.

To avoid this problem CACAO uses a *safe stack* area during replacement: Before the new execution state is created, CACAO allocates a memory block to serve as a temporary stack area. All necessary info for rebuilding the execution state is copied from local variables (which are unsafe) to this memory block. Then a function of the architecture layer is called. This function switches to the safe stack area and calls the function for building the execution state. Afterwards, the stack is switched back to the normal stack area of the thread—which now contains the new execution state—and the safe stack area is freed.

Algorithm 2 Algorithm for recovering the source state

Input:
E: execution state
P: initial replacement point
C: codeinfo containing P
I: innermost stack frame info of current thread
Output:
E: residual execution state
S: source state

Algorithm:
$S \leftarrow ()$
while $(P \neq \text{NULL}) \vee (I \neq \text{NULL})$:
  if $P = \text{NULL}$:
    $(E, F) \leftarrow$ pop-native-frame(E, I)
    $F \leftarrow F : S$
    $I \leftarrow I$`->prev`
    if in-java-code(E):
      $C \leftarrow$ get-code(E)
      $P \leftarrow$ find-replacement-point(E, C)
    end if
  else
    $F \leftarrow$ read-source-frame(E, C, P)
    $S \leftarrow F : S$
    if P is within an inlined body:
      $P \leftarrow$ find-inline-start(C, P)
    else
      $C' \leftarrow$ find-caller(E, C)
      if no caller was found:
        return (S, E)
      $E \leftarrow$ pop-activation-record(E, C)
      $C \leftarrow C'$
      $P \leftarrow$ find-replacement-point(E, C)
    end if
  end if
end while
return (S, E)

Algorithm 3 Algorithm for rebuilding the execution state

Input:
E: residual execution state
S: source state
Output:
E: execution state
P: target replacement point

Algorithm:
$P \leftarrow$ NULL
$P_{\text{parent}} \leftarrow$ NULL
while $S \neq ()$:
  $F \leftarrow \text{head}(S)$
  $S \leftarrow \text{tail}(S)$
  if F is a native frame:
    $E \leftarrow$ push-native-frame(E, F)
    $P \leftarrow$ NULL
    $P_{\text{parent}} \leftarrow$ NULL
  else
    if $P_{\text{parent}} =$ NULL :
      $E \leftarrow$ push-activation-record(E, F, P)
    end if
    $P \leftarrow$ get-target-replacement-point(F)
    $E \leftarrow$ write-source-frame(E, F, P)
    if P is at an inlined call site:
      $P_{\text{parent}} \leftarrow P$
    else
      $P_{\text{parent}} \leftarrow$ NULL
    end if
  end if
end while
return (E, P)

Chapter 7

Testing

Recompilation and adaptive inlining are designed to be applied to the relatively small set of hot methods of a program. Thus it is hard for regression tests to cover significant parts of the involved code. This chapter briefly discusses how CACAO's inlining and replacement code have been tested.

7.1 Testing Inlining

In order to independently test method inlining, we would like to have special test configurations:

- A configuration that separates inlining from recompilation. Inlining will be applied for all compiled methods in this case.
- A configuration in which inlining does not need on-stack replacement. In contrast to the previous item this requires that either no speculative inlining is done, or the test cases are selected such that no deoptimization will be necessary.

In order to provide these configurations, debug builds of CACAO have the following command line options:

-ia inline transform *all* compiled methods,

-ip be *pessimistic*, i.e. do not apply speculative inlining

-ix apply speculative inlining, but abort with an error message if an assumption is broken, instead of triggering recompilation.

As CACAO includes a type-inferring verifier that operates on the intermediate representation, the verifier can be used to check the results of the inlining transformation: During development a slightly modified version of the verifier was run over the IR after inlining. In this way, bugs causing the inliner to produce code that violates type-safety were reliably exposed.

7.2 Debugging the Inlining Transformation

For debugging purposes it is of great advantage to narrow down the set of transformed methods. The difficulty of finding a compiler bug usually rises with the size of the compilation unit exposing the bug. Thus a development option was built into CACAO to only use the result of the inlining transformation if it is below a certain number of intermediate instructions. By running large test cases while gradually increasing this threshold, it was possible to observe most initial bugs of the inlining transformation in the smallest case that triggered them. This greatly simplified the debugging process.

7.3 Testing On-Stack Replacement

For testing on-stack replacement, the compiler was configured to activate all replacement traps in a method as soon as it has been compiled and to recompile methods using the baseline compiler when a replacement trap is triggered. In this way every called method is replaced at least once. Most methods are replaced more often, as they are on the stack when methods called by them trigger their replacement traps.

One problem with this testing scheme is that source and target code of a replacement typically have identical stack layout and register allocation (as they are both produced by the baseline compiler, only at different times during program execution). To make the tests more significant, code was written to overwrite stack contents and registers in the execution state between recovering the source state and building the new execution state. This way the new execution state only contains values that were translated by the replacement system, and none that accidentally remained in place during the translation.

Chapter 8

Results

This chapter reports experimental results obtained with CACAO and the SPECjvm98 benchmark suite using adaptive inlining.

8.1 Comparison of Inlining Heuristics

The programs of the SPECjvm98 benchmark suite were executed with recompilation triggered by countdown traps. Hot methods were recompiled with inlining and without instrumentation code. The benchmarks were also run with recompilation done by the baseline compiler (no inlining). The purpose of the latter runs, in which recompilation was only used to remove instrumentation code, was to separately measure the overhead introduced by the countdown traps (see Table 5).

Tables 5, 6, 7, and 8 summarize the results[1]. Times are the minimum execution times[2] of 20 runs each. The standard deviation of each sample is given in parentheses.

Figure 7 visualizes the relative execution time of each benchmark program for all heuristics.

8.1.1 Aggressive Depth-first Inlining

As Table 5 shows, depth-first heuristics performed well for `compress` and `mtrt`, improving execution time by 8.6% and 15% respectively. However, for `mpegaudio` performance got slightly worse, although over 80% of executed method calls were eliminated (see Table 9). In this case, side effects of

[1]Note that these benchmark runs were *not* executed according to the SPECjvm98 specifications and thus do not represent valid SPEC results.

[2]Real-time measurements were used rather than user time, as—contrary to expectations—elapsed real-time proved to be statistically much more stable on the system used. Examination of the reported user and system time values suggests that the system tends to overestimate system time in some cases which leads to outliers with artificially reduced user time.

inlining seem to degrade code quality. The large number of eliminated calls, however, indicates that even in this case the potential benefits of inlining could be great. Better performance may be expected when the linear scan register allocator and full copy elimination are available in CACAO.

The downside of aggressive depth-first inlining is severe expansion of compiled code size (see Table 13). Thus a version of depth-first inlining was implemented that completely cancels inlining for a root method if the code expansion threshold is reached. (The rationale being that partial depth-first inlining yields very unbalanced inlining trees, so one could expect its benefits to be rather small, anyway.) As to be expected, this change reduces code expansion significantly (see Table 14). The modified algorithm, however, yields bad results for two benchmarks. In the case of `javac` execution time increases by significant five percents compared to the unoptimized case. Clearly, a more sophisticated solution is needed to limit code expansion.

8.1.2 Aggressive Breadth-first Inlining

Aggressive breadth-first inlining was implemented in order to obtain more balanced inlining trees than those built by depth-first inlining. For the results presented in Table 7, expansion of intermediate code was limited to a factor of five. This translates to a maximal expansion of compiled code by approximately the same factor (Table 15 shows a maximum factor of 4.6 to 8.3 over all methods). For some benchmarks, breadth-first inlining yielded performance better than or comparable to depth-first inlining with less code expansion. Total code expansion is still high, though, especially for `javac`, which also becomes slower than the baseline version by over 4%. So breadth-first inlining is not generally superior to depth-first inlining.

8.1.3 Knapsack Heuristics

The knapsack algorithm was the only algorithm that improved execution time for all benchmarks in the SPECjvm98 suite. As can be seen in Table 8, the variance of the achieved speedups is very large, with improvements ranging from 0.8% to 18.2%. The knapsack heuristics eliminated on average over 72% of executed method calls (somewhat less than aggressive depth-first inlining which eliminated over 76%). In the worst case still more than 46% of calls could be eliminated. This indicates that we may expect further performance improvements with an improved compiler back-end using linear scan register allocation and copy elimination.

8.2 Number of Executed Method Calls

Figure 8 shows how effective the various heuristics were in reducing the dynamic number of executed calls for each benchmark. Tables 9, 10, 11,

benchmark	baseline	inlining	change[3]	instrumentation	change[3]
compress	8.29 (0.05)	7.58 (0.05)	-8.6%	8.31 (0.02)	+0.2%
jess	7.06 (0.03)	7.01 (0.05)	-0.7%	7.12 (0.02)	+0.8%
db	17.77 (0.02)	17.12 (0.03)	-3.7%	17.79 (0.03)	+0.1%
javac	10.54 (0.09)	10.11 (0.49)	-4.1%	10.77 (0.08)	+2.2%
mpegaudio	8.97 (0.04)	9.05 (0.07)	+0.9%	9.00 (0.02)	+0.3%
mtrt	6.14 (0.03)	5.19 (0.04)	-15 %	6.15 (0.09)	+0.2%
jack	7.09 (0.04)	6.94 (0.02)	-2.1%	7.20 (0.01)	+1.6%

Table 5: Benchmark execution time results, depth-first heuristics, all times in seconds

benchmark	baseline	adaptive inlining	change[4]
compress	8.29 (0.05)	7.60 (0.04)	-8.3%
jess	7.06 (0.03)	7.08 (0.04)	+0.3%
db	17.77 (0.02)	17.34 (0.06)	-2.4%
javac	10.54 (0.09)	11.08 (0.23)	+5.1%
mpegaudio	8.97 (0.04)	8.92 (0.05)	-0.6%
mtrt	6.14 (0.03)	5.27 (0.04)	-14 %
jack	7.09 (0.04)	6.96 (0.09)	-1.8%

Table 6: Benchmark execution time results, depth-first heuristics with cancelling, all times in seconds

and 12 contain the detailed results. The second column of each table gives the number of calls performed without any optimization, the third column gives the number of calls performed with adaptive inlining. The numbers show that adaptive inlining could in all cases reduce the number of calls significantly. In the case of **compress**, only 0.05% of the original number of calls was performed when using aggressive depth-first inlining.

8.3 Code Size

For each recompiled method the change of code size relative to the code generated by the baseline compiler was measured. Tables 13, 14, 15, and 16 show the frequency distribution of the expansion factor, and its geometric mean and maximum. Note that these factors refer to individual recompiled methods and not to total code size change. An interesting observation is that in many cases, inlining does not increase the size of the compiled code at all, and quite often even reduces it. The reason for this can probably be found in the object-oriented programming style used with Java that favors very small methods, for example getter/setter methods containing a single statement, and empty initializers. Code size expansion—while for some benchmarks

benchmark	baseline	adaptive inlining	change[5]
compress	8.29 (0.05)	7.63 (0.06)	-8.0%
jess	7.06 (0.03)	6.82 (0.10)	-3.4%
db	17.77 (0.02)	17.10 (0.13)	-3.8%
javac	10.54 (0.09)	10.99 (0.41)	+4.3%
mpegaudio	8.97 (0.04)	8.93 (0.09)	-0.4%
mtrt	6.14 (0.03)	5.18 (0.07)	-16 %
jack	7.09 (0.04)	6.99 (0.07)	-1.4%

Table 7: Benchmark execution time results, breadth-first heuristics, all times in seconds

benchmark	baseline	adaptive inlining	change[6]
compress	8.29 (0.05)	7.74 (0.02)	-6.6%
jess	7.06 (0.03)	6.77 (0.04)	-4.1%
db	17.77 (0.02)	17.21 (0.02)	-3.2%
javac	10.54 (0.09)	10.35 (0.17)	-1.8%
mpegaudio	8.97 (0.04)	8.84 (0.04)	-0.8%
mtrt	6.14 (0.03)	5.02 (0.03)	-18.2%
jack	7.09 (0.04)	6.77 (0.01)	-4.5%

Table 8: Benchmark execution time results, knapsack heuristics, all times in seconds

and heuristics rarer than reduction—still dominates on the whole.

If support for releasing unused compiled code would be implemented, the code expansion factors could effectively become smaller, because in cases where all calls to a method get inlined, the compilation unit for this root method could be released.

Tables 13, 14, 15, and 16 also display total code size change for each benchmark. These numbers take into account that memory used by obsolete code is currently not freed in CACAO, and so both the original instrumented compilation unit and the recompiled code are included in the total. Thus total code size changes are much higher than the code expansion caused by inlining itself.

8.4 Recompilations

The number of recompilations was counted for each benchmark. Tables 17, 18, 19, and 20 list the total number of methods compiled by the baseline compiler for each benchmark and the number of recompilations. The baseline numbers vary slightly between different heuristics, as in some cases

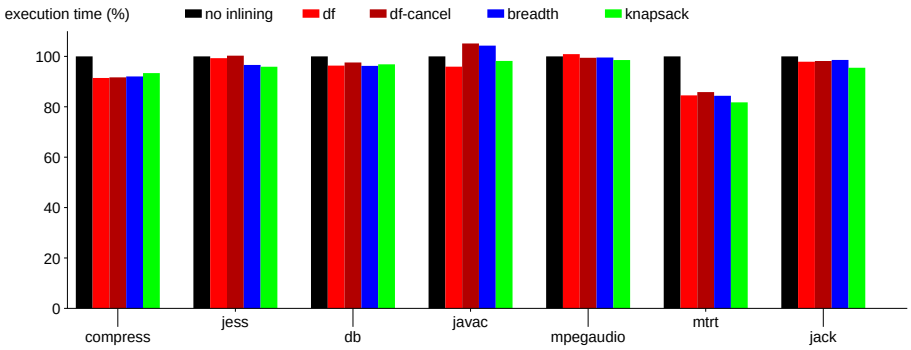

Figure 7: Relative execution times with inlining

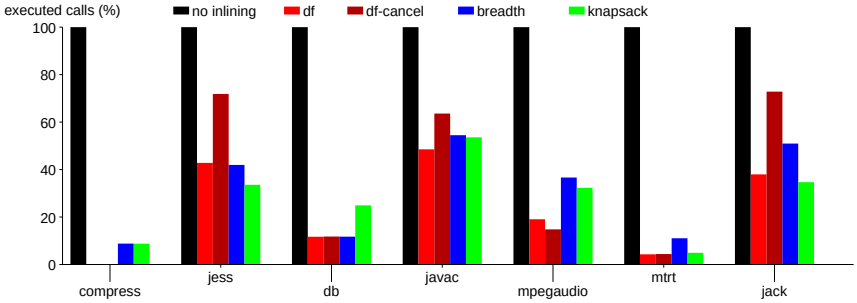

Figure 8: Relative number of executed calls

some methods are only ever executed inline and thus are “hidden” from the baseline compiler.

benchmark	calls (unopt)	calls (opt)	change
compress	226048394	80640	-99.96%
jess	122933993	52627260	-57.2%
db	169932836	19858738	-88.3%
javac	114616271	55612081	-51.5%
mpegaudio	109847865	20932623	-80.9%
mtrt	287453574	12301007	-95.7%
jack	100086803	38001762	-62.0%

Table 9: Number of calls executed, depth-first heuristic

benchmark	calls (unopt)	calls (opt)	change
compress	226048394	94151	-99.95%
jess	122933993	88324901	-28.2%
db	169932836	20019251	-88.2%
javac	114616271	72892305	-36.4%
mpegaudio	109847865	16260550	-85.2%
mtrt	287453574	12665017	-95.6%
jack	100086803	46260565	-53.8%

Table 10: Number of calls executed, depth-first heuristic with cancelling

benchmark	calls (unopt)	calls (opt)	change
compress	226048394	19894348	-91.2%
jess	122933993	51566905	-58.1%
db	169932836	19927446	-88.3%
javac	114616271	62437784	-45.5%
mpegaudio	109847865	40245402	-63.4%
mtrt	287453574	31774462	-88.9%
jack	100086803	50991866	-49.1%

Table 11: Number of calls executed, breadth-first heuristic

benchmark	calls (unopt)	calls (opt)	change
compress	226048394	19820463	-91.2%
jess	122933993	41290870	-66.4%
db	169932836	42328511	-75.1%
javac	114616271	61399060	-46.4%
mpegaudio	109847865	35472405	-67.7%
mtrt	287453574	14139733	-95.1%
jack	100086803	34712346	-65.3%

Table 12: Number of calls executed, knapsack heuristic

factor	compress	jess	db	javac	mpegaudio	mtrt	jack
≤ 0.2	0.0%	1.5%	1.1%	0.6%	0.0%	0.0%	0.8%
≤ 0.5	0.0%	1.5%	1.1%	5.7%	0.0%	4.6%	1.5%
≤ 1.0	38.1%	32.3%	37.6%	38.9%	41.1%	47.7%	30.4%
≤ 2.0	14.3%	16.2%	12.9%	16.5%	17.7%	13.8%	10.3%
≤ 5.0	19.0%	17.2%	25.8%	23.9%	17.7%	16.2%	27.8%
≤ 10.0	23.8%	21.2%	16.1%	11.0%	13.5%	11.5%	20.2%
> 10.0	4.8%	10.1%	5.4%	3.4%	9.9%	6.2%	9.1%
geometric mean	2.3	2.3	2.0	1.6	2.1	1.6	2.5
maximal	13.3	25.4	19.0	20.7	100.5	24.2	19.9
total size (unopt)	325631	430249	338241	622537	641378	384626	497758
total size (opt)	493914	1076228	534004	1811068	978867	803793	1136084
change	+52%	+139%	+58%	+191%	+53%	+109%	+128%

Table 13: Code size change through inlining and recompilation, depth-first heuristics

factor	compress	jess	db	javac	mpegaudio	mtrt	jack
≤ 0.2	0.0%	1.0%	1.1%	0.6%	0.0%	0.0%	1.2%
≤ 0.5	0.0%	2.0%	1.1%	5.7%	0.0%	4.3%	2.0%
≤ 1.0	53.5%	66.0%	48.4%	52.3%	48.2%	56.5%	55.1%
≤ 2.0	9.3%	9.4%	16.5%	15.2%	19.9%	14.5%	7.5%
≤ 5.0	16.3%	9.4%	18.7%	18.7%	14.2%	11.6%	16.9%
≤ 10.0	18.6%	9.4%	11.0%	6.6%	10.6%	8.0%	12.2%
> 10.0	2.3%	3.0%	3.3%	0.9%	7.1%	5.1%	5.1%
geometric mean	1.8	1.3	1.5	1.3	1.8	1.3	1.6
maximal	10.1	17.6	13.4	15.7	100.9	24.4	19.6
total size (unopt)	325631	430249	338241	623335	653269	384988	498192
total size (opt)	412478	646024	430261	1301556	888638	686833	807462
change	+27%	+50%	+27%	+109%	+36%	+78%	+62%

Table 14: Code size change through inlining and recompilation, depth-first heuristics with cancelling

factor	compress	jess	db	javac	mpegaudio	mtrt	jack
≤ 0.2	0.0%	0.5%	0.0%	0.9%	0.0%	0.0%	0.8%
≤ 0.5	0.0%	2.5%	2.0%	5.7%	2.7%	3.8%	2.3%
≤ 1.0	40.5%	34.8%	36.4%	36.3%	43.8%	51.9%	32.4%
≤ 2.0	9.5%	15.2%	18.2%	22.1%	19.9%	14.7%	17.2%
≤ 5.0	50.0%	43.9%	41.4%	33.7%	32.9%	27.6%	45.0%
≤ 10.0	0.0%	3.0%	2.0%	1.4%	0.7%	1.9%	2.3%
> 10.0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
geometric mean	1.8	1.7	1.6	1.4	1.4	1.2	1.7
maximal	4.6	5.5	5.5	8.3	6.3	5.5	6.1
total size (unopt)	327371	429822	338382	620997	653582	385209	498691
total size (opt)	410809	712198	443547	1403373	824384	603767	912318
change	+26%	+66%	+31%	+126%	+26%	+57%	+83%

Table 15: Code size change through inlining and recompilation, breadth-first heuristics

factor	compress	jess	db	javac	mpegaudio	mtrt	jack
≤ 0.2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
≤ 0.5	0.0%	1.1%	0.0%	5.9%	0.0%	0.9%	2.0%
≤ 1.0	29.3%	23.2%	26.2%	29.4%	44.0%	35.3%	25.5%
≤ 2.0	41.5%	24.7%	31.0%	31.0%	32.0%	37.9%	25.5%
≤ 5.0	29.3%	40.5%	34.5%	27.9%	21.3%	22.4%	38.7%
≤ 10.0	0.0%	8.9%	6.0%	4.9%	1.3%	2.6%	5.9%
> 10.0	0.0%	1.6%	2.4%	0.9%	1.3%	0.9%	2.5%
geometric mean	1.5	2.0	1.8	1.5	1.4	1.5	1.9
maximal	3.7	24.1	12.5	16.8	13.0	12.2	20.0
total size (unopt)	327376	429301	338273	620963	653582	385663	498019
total size (opt)	375996	640347	422720	1163337	794648	534157	802259
change	+15%	+49%	+25%	+87%	+22%	+39%	+61%

Table 16: Code size change through inlining and recompilation, knapsack heuristics

benchmark	methods	recompiled	replacements
compress	769	42 (6%)	61
jess	1250	198 (16%)	310
db	804	93 (12%)	141
javac	1574	653 (42%)	1363
mpegaudio	947	141 (15%)	234
mtrt	908	130 (14%)	201
jack	1019	263 (26%)	549

Table 17: Number of recompilations, depth-first heuristics

benchmark	methods	recompiled	replacements
compress	773	45 (6%)	67
jess	1254	208 (17%)	311
db	805	99 (12%)	152
javac	1586	670 (42%)	1382
mpegaudio	950	142 (15%)	239
mtrt	932	138 (15%)	217
jack	1025	261 (25%)	521

Table 18: Number of recompilations, depth-first heuristics with cancelling

benchmark	methods	recompiled	replacements
compress	772	42 (5.4%)	61
jess	1246	198 (16%)	321
db	803	99 (12%)	159
javac	1571	662 (42%)	1412
mpegaudio	949	146 (15%)	232
mtrt	920	156 (17%)	259
jack	1024	262 (26%)	503

Table 19: Number of recompilations, breadth-first heuristics

benchmark	methods	recompiled	replacements
compress	770	41 (5%)	59
jess	1240	190 (15%)	328
db	802	84 (11%)	142
javac	1534	555 (36%)	1170
mpegaudio	949	150 (16%)	246
mtrt	905	116 (13%)	214
jack	1016	204 (20%)	467

Table 20: Number of recompilations, knapsack heuristics

Chapter 9

Summary

Current virtual machines use dynamic compilation and adaptive optimization in order to deliver high performance while keeping compilation times low. A framework for adaptive optimization has been implemented in the CACAO virtual machine, using instrumentation and on-stack replacement.

The baseline compiler of CACAO instruments code with counters that trigger the recompilation of hot methods. On-stack replacement is then used to install optimized versions of hot methods. If assumptions made by the optimizer are broken by dynamic class loading, on-stack replacement is invoked again to undo optimizations that have become invalid.

Method inlining has been implemented as a pass in the optimizing compiler. The inliner supports several heuristics for making inlining plans. Of the implemented heuristics, a variant of the greedy knapsack algorithm proved to yield the best overall performance. Improvements of execution time up to 18% were achieved for the SPECjvm98 benchmark suite. As adaptive inlining could on average eliminate over 70% of executed calls, further improvements can be expected when the linear scan register allocator is available in CACAO.

An algorithm for eliminating local subroutines has been presented. The algorithm works by representing return addresses as types and specializing basic blocks with respect to these types. All uses of `JSR` and `RET` instructions that conform to the JVM specification can be handled.

On-stack replacement has been implemented in CACAO in order to allow replacing the code of active methods. The replacement mechanism can switch between unoptimized and optimized code in both directions and between different optimized versions. Stack frames can be combined or split, as replacement translates between inlined and non-inlined method calls.

Promising directions for future work are: guiding inlining with more precise profiling data, using inter-procedural type analysis, and creating replacement points on demand to reduce memory consumption.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974.

[2] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000.

[4] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 52–64, New York, NY, USA, 2000. ACM Press.

[5] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. *Lecture Notes in Computer Science*, 2374:498–524, 2002.

[6] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 134–145, New York, NY, USA, 1997. ACM Press.

[7] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[8] John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, 1992.

[10] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[11] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *PLDI*, pages 85–94, 1988.

[12] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency - Practice and Experience*, 16(7):647–670, 2004.

[13] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: a case study. *ACM Lett. Program. Lang. Syst.*, 1(1):22–32, 1992.

[14] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[15] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 273–282, New York, NY, USA, 1994. ACM Press.

[16] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[17] Jeffrey A. Dean. *Whole-program optimization of object-oriented languages*. PhD thesis, University of Washington, 1996.

[18] David Detlefs and Ole Agesen. Inlining of virtual methods. *Lecture Notes in Computer Science*, 1628:258–277, 1999.

[19] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Stephen N. Freund. The costs and benefits of Java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[22] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, New York, NY, USA, 1992. ACM Press.

[23] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 187–204, New York, NY, USA, 2003. ACM Press.

[24] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland.

[25] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[26] Junpyo Lee, Byung-Sun Yang, Suhyun Kim, Kemal Ebcioğlu, Erik Altman, Seungil Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, and Soo-Mook Moon. Reducing virtual call overheads in a Java VM just-in-time compiler. *SIGARCH Comput. Archit. News*, 28(1):21–33, 2000.

[27] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[28] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot$^{\text{TM}}$ server compiler. In *Java^{TM} Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[29] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 21, 1999.

[30] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, 1977.

[31] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th*

ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pages 180–195, New York, NY, USA, 2001. ACM Press.

[32] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed calls in Java packages. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–92, New York, NY, USA, 2000. ACM Press.