FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimization Framework for the CACAO VM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Josef Eisl
Matrikelnummer 0625147

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 03.12.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimization Framework for the CACAO VM

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Josef Eisl
Registration Number 0625147

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 03.12.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Josef Eisl
Schafgasse 25, 5017 Wals

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First, I want to thank my parents for their continuous and invaluable support and for believing in me throughout my whole life. There are no words to say how grateful I am.

Thanks to Prof. Andreas Krall for sharing his extensive knowledge in the field of compiler construction, computer languages and virtual machines and for giving me the opportunity to work on the CACAO VM.

I want to thank all people involved with the CACAO development, especially Stefan Ring for giving me the chance to realize my ideas regarding the CACAO project.

A very special thanks to my friends Bernhard Urban and Harald Steinlechner for numerous, sometimes heated, debates on computer languages, compilers, virtual machines and the universe in general. Also for their motivation, their constructive comments on this work and for one or another pint of beer.

Another big thank you to Clemens Horak for proof reading and for many style improvements.

Last but definitely not least I want to thank Marianne for her unlimited support and for bearing my ups and downs during the last months in which this work developed. It would not have been possible without you!

Abstract

A virtual machine is a software that takes charge of executing another program. This either can be done by *interpreting* the instructions of the program, or by *compiling* parts of it when needed. The second approach is also called *just-in-time* (JIT) compilation.

The CACAO VM is a virtual machine for Java bytecode. It follows a JIT-only approach meaning that all methods are compiled into native machine code prior execution. This requires a fast compiler to minimize the latency for program execution. Consequently, the compiler, called *baseline* compiler, uses simple data representation and the passes are highly integrated. While the generated machine code is adequate for most parts of a program, the over-all performance can be improved, if more time for better optimization is invested for frequently executed methods. It can not be decided a priori which methods will be called regularly. To gain this knowledge the virtual machine profiles the run-time behavior of the program and selects methods for recompilation with more efforts. This approach is known as *adaptive optimization*.

So far the CACAO VM uses the baseline compiler with a higher optimization level for recompilation. This approach has two problems. On the one hand, the additional optimizations complicate the baseline compiler and maintenance is more difficult. On the other hand and more important, due to the simple but efficient construction the baseline compiler is inflexible and new optimizations are hard to implement.

This work presents a new compilation framework for CACAO, which replaces the baseline compiler as the optimizing compiler. It features two intermediate representations. The graph-based *high-level representation* is intended for program analysis and transformation. It is designed to make optimization development fast and easy. The *low-level representation* is focused on tasks close to the machine level, like register allocation and code emission. The compiler contains a *pass manager* to administer the execution and data exchange of passes. Currently, the transformation pipeline includes IR construction, structural analysis, scheduling, instruction selection, register allocation and code emission.

An empirical comparison between the baseline compiler and the new framework discloses the potential of the new framework and sets the agenda for future directions of the project.

Kurzfassung

Eine Virtuelle Maschine (VM) ist eine Software, die verwendet wird, um andere Programme auszuführen. Das geschieht entweder durch *Interpretieren* der einzelnen Instruktionen oder durch *Übersetzen* von Programmteilen in Maschinencode. Da beim zweiten Ansatz die Übersetzung erst stattfindet, wenn das Programm benötigt wird, spricht man auch von *just-in-time* (JIT) Übersetzung.

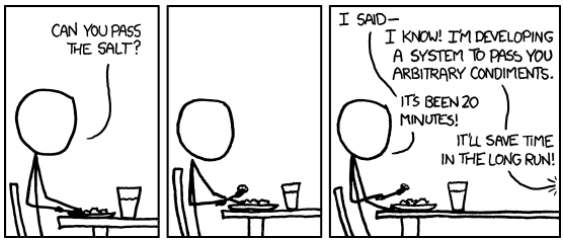
Die CACAO VM ist eine virtuelle Maschine für Java Bytecode. CACAO ist eine reine JIT VM. Das bedeutet, alle Methoden werden übersetzt, bevor sie ausgeführt werden. Dazu bedarf es eines schnellen Übersetzers um eine zeitnahe Programmausführung zu gewährleisten. Dazu verwendet der sogenannte *Baseline Compiler* einfache Datenstrukturen und die einzelnen Übersetzungsschritte sind stark an einander gekoppelt. Für die meisten Programmteile reicht das Ergebnis des Baseline Compilers aus. Für häufig verwendete Methoden würde es sich allerdings auszahlen, mehr Zeit in eine bessere Übersetzung zu investieren. Da man im Vorhinein nicht feststellen kann, welche Programmteile oft verwendet werden, untersucht die virtuelle Maschine das Laufzeitverhalten der Applikation und wählt darauf aufbauend Methoden für erneute Übersetzung aus. Dieses Verfahren wird *adaptive Optimierung* genannt.

Bisweilen verwendet CACAO den Baseline Compiler für die Optimierung häufig genutzter Teile. Das hat mehrere Nachteile. Zum Einen wird der Baseline Compiler dadurch immer komplizierter und die Wartung aufwendiger. Das größere Problem liegt allerdings darin, dass der Baseline Compiler durch die einfache, aber effiziente Form unflexibel ist und sich das Entwickeln neuen Optimierungen schwierig gestaltet.

Diese Arbeit beschreibt ein neues Übersetzer-Framework, das den Baseline Compiler als optimierenden Übersetzer ablösen soll. Das Framework verwendet zwei neue Zwischendarstellungen. Die graph-basierte *high-level* Darstellung eignet sich besonders für Programmanalysen und Transformationen. Ihr Ziel ist es den Entwicklungsprozess für Optimierungen einfach und schnell zu gestalten. Die *low-level* Darstellung ist speziell an die Bedürfnisse von maschinennahen Aufgaben, wie zum Beispiel Register Allokation oder der Generierung von Maschinencode, angepasst. Der Übersetzer verfügt über einen *Pass Manager*, der sich um die Ausführung und den Datenaustausch der Pässe kümmert. Zur Zeit beinhaltet die Übersetzer-Pipeline die IR-Graphen Erstellung, strukturelle Analysen, verschiedene Scheduler, Befehlsauswahl, einen Register Allokator sowie einen Maschinencode Emitter.

Der empirische Vergleich mit dem Baseline Compiler zeigt das Potential des neuen Systems und gibt zukünftige Aufgaben vor.

Contents



I find that when someone's taking time to do something right in the present, they're a perfectionist with no ability to prioritize, whereas when someone took time to do something right in the past, they're a master artisan of great foresight.

CHAPTER 1

Introduction

1.1 Virtual Machines

A classic compiler translates a high-level language, for instance C or C++, into a binary format the processor can execute. In general this binary image is only usable on the machine it was compiled for. To run the program on another architecture it has to be recompiled. If there are P programs and M machines then $P \times M$ different executable images exist. To conquer this multiplicative explosion an intermediate format between the high-level language and the binary code is introduced. Rau (1978) identified the following categories for program representation:

- **HLR:** high-level language representation (C, C++, Java)
- **DIR:** direct interpretable representation (Java bytecode[1], CLI[2])
- **DER:** direct executable representation (binary code)

Instead of translating from HLR to DER the compiler emits a program in DIR format. This DIR is platform independent and can be shared between different computers. There are two ways of executing a DIR (also called *bytecode*) image. The first one is interpretation. That means the instructions are simulated by an interpreter. The second way is to translate the bytecode into a DER. This translation can happen before the program is executed (*ahead-of-time*) or on demand (*just-in-time* or JIT). The later approach has the advantage that the compiler only translates the parts of the program that are executed. Additionally, the program can add new DIR code at run-time which is not possible with an ahead-of-time compiler.

A software that manages the execution of DIR images is called a *virtual machine*. This setup changes the multiplication in the equation above to an addition. Maintaining P programs on M physical machines requires P DIR programs and M virtual machines which sums up to $P + M$

[1] Lindholm et al. (2013)

[2] Common Language Infrastructure (ISO, 2012a)

images in total. Also, adding a new program means only to compile it once into the DIR format, no matter how many physical machines are supported. Similarly, adding a new machine does not require recompiling any of the programs.

Program execution is not the only task of a virtual machine. It often also provides run-time components, for instance a garbage collector, a standard library or a dynamic loader and linker. A virtual machine can also increase security by restricting the power of the DIR and verifying images prior to execution.

The virtual machine approach is very popular with recent language implementations. Examples include Java, Python, Microsoft .NET languages, JavaScript or Ruby to name just a few. Some of these examples also show that having a DIR is not mandatory. If the source code is distributed, the compiler is part of the virtual machine.

1.2 The CACAO Virtual Machine

The CACAO VM is a virtual machine for Java bytecode. Krall and Grafl (1997) introduced it as a fast just-in-time VM for the DEC Alpha processor. Since then CACAO was expanded in terms of feature completeness as well as architecture support. Currently the CACAO VM actively maintains backends for the Alpha, ARM32, MIPS, PowerPC, PowerPC64 and x86/x86_64 architecture. CACAO runs on all modern Unix/Linux based operating systems. GNU Classpath, OpenJDK6 or OpenJDK7 are supported Java Class Libraries. CACAO is integrated into the IcedTea build project for OpenJDK.[3]

CACAO follows a compile-only approach which means that it compiles every method to native machine code before the method is executed. This is in contrast to systems with interpreters where executions starts in the interpretation mode and compilation is only triggered for frequently executed parts of the program.

1.3 Motivation

Despite the fact that CACAO always was, and hopefully will always be, a playground for research and education, it also strives to be a standard compliant JVM implementation (Thalinger, 2004). Unfortunately this comes at the cost of high complexity which in turn leads to a flat learning curve for new participants. The goal of this work is to abstract some of the complexity and to provide an easy to use framework for implementing new optimization and analysis passes. It should help to make CACAO more attractive for students and researchers.

1.4 Problem Statement

Because of CACAOs compile-only approach compile time is important. That dictates the use of fast algorithms and simple data structures. A so called *baseline* compiler produces acceptable code in short time. This is sufficient for most parts of the program but for frequently executed

[3] `http://icedtea.classpath.org`

(*hot*) methods, more aggressive and time consuming optimization would be desirable. Unfortunately the compiler does not know whether a method is hot or not at the time of the first compilation. The virtual machine gathers this knowledge during the lifetime of a program. It inspects the run-time behavior and recompiles frequently used methods using an *optimizing compiler*. The details of this technique, called *adaptive optimization*, are the topic of Section 2.2.

Until now the CACAO VM performed this recompilation by calling the baseline compiler with a different set of optimizations. While maintaining only one compiler has its advantages, there are also issues with this approach. First, the baseline compiler is a highly tuned piece of software. All passes interact with each other on multiple levels which are opaque to the developer. Creating new optimizations requires a deep knowledge about the compiler and the interdependencies between passes. The second problem with the single-compiler approach is that changes always affect the baseline compiler. Some passes can be deactivated by a simple if-statement but as soon as central compiler data structures are extended, the baseline compiler performance degrades. Finally, every additional pass adds new dependencies and the compiler gets more complicated.

1.5 Aim of the Work

The aim of this work is to tackle the problems described above by creating a dedicated optimizing compiler. This means designing and implementing a compiler framework which is easy to use and decreases the workload when creating new analysis and optimization passes. The passes exchange information via a clear interface. Interdependencies and assumptions are visible to the developer.

Optimizer passes can be moved to the new infrastructure and removed from the original baseline compiler. Consequently, the code is easier to understand and the overall complexity decreases.

1.6 Methodological Approach

The new compiler (*second-stage compiler*) is a drop-in replacement for the old (from now on called *legacy*) optimizing compiler. The following questions occur:

- What are the inputs and what is the expected result of the compiler?
- How will the baseline compiler and the second-stage compiler interact?
- What are the interfaces to the virtual machine's run-time?

The new compiler framework is created from scratch. To avoid unnecessary design constraints components from the legacy compiler are not reused. No parts from the existing compiler are touched.

A self-imposed restriction is to avoid additional dependencies on external libraries. The first reason for this is that libraries might not be available on all architectures and operating systems that are supported by CACAO. Also, using external software always means a loss of influence.

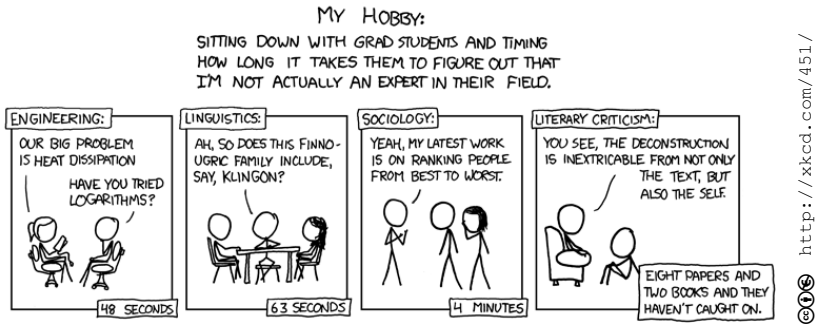
Experience has shown that pushing contributions back to upstream projects can pose a major burden. Finally, there are legal concerns when using third-party software. A number of GNU Linux distributions contain CACAO so this issue is not insignificant.

1.7 Organization of the Work

The rest of this work is organized as follows: The next chapter (Chapter 2) gives an overview of the state of the art in the field of virtual machines and adaptive optimization. It is the justification for the design decisions made throughout this work and is therefore covered in more detail.

Chapter 3 discusses the implementation of the compiler framework. It starts with a description of the main data structures, the high-level and the low-level intermediate representation. The subsequent part outlines pass pipeline and other supporting components of the framework. The following chapter (Chapter 4) describes the compiler passes that were implemented in course of this work. Chapter 5 presents the empirical evaluation and provides an comparison between the existing and the new compiler.

Chapter 6 evaluates the design from a critical point of view and discusses suboptimal decisions and problems. It also proposes possible solutions to these issues. The final chapter (Chapter 7) outlines future perspectives and ideas for enhancing the framework. It also discusses open problems of the implementation. The chapter concludes with a short summary of the work.



If you think this is too hard on literary criticism, read the Wikipedia article on deconstruction.

CHAPTER 2

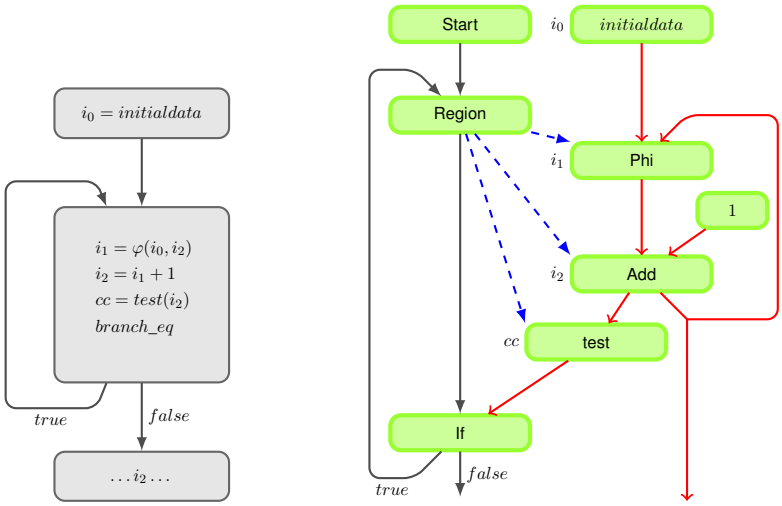
State of the Art

2.1 Intermediate Representation

The intermediate representation (IR) is one of the core components of a compiler. Click and Paleczny (1995) concluded that it is vital that the transformations and analyses executed on top of this IR are 1.) fast and correct and 2.) simple to write, understand and extend. To meet these requirement Click and Paleczny developed a novel graph-based IR.

In the classic model a list of instructions forms basic blocks with a single entry and a single exit. These basic blocks are the nodes of the control-flow graph (CFG). Variables or virtual registers transfer the data from one instruction to another. The basic idea of Click and Paleczny is to dismiss the burden of preserving a linear schedule at all times. Their IR stores only the information necessary to preserve the semantics of the original program. In addition to the control-flow graph, data dependencies in the form of a data-flow graph (DFG) are required. Click and Paleczny merge both graphs into a single representation. The vertices of this graph are either primitive nodes, the instructions, or special control-flow nodes which replace the basic blocks in traditional intermediate representations. Two kinds of edges are used to describe the semantics of a program. *Control edges* model control transfers in the CFG, whereas *data edges* describe data dependencies from the DFG.

They use *Static Single Assignment* (Rastello, 2013; Cytron et al., 1991) form which guarantees that there is only one definition of a value. This renders the need of variables to pass on values unnecessary. The input operands of instructions are simply pointers to the instruction defining the value. These pointers, also called *use-def* edges, describe the data-flow in the graph. The control-flow is modeled using special nodes called Region. The input of a region is an ordered list of control inputs. The only control output is used to define the successor. Control-flow splits are achieved by If nodes. They have a control input and a data input, the *predicate*, which is used to choose the correct successor. Primitive instructions have a control input to express



(a) Basic Block IR (b) Graph IR

Figure 2.2: Comparison of Intermediate Representations. — Black edges symbolize control-flow. Data-dependencies (def-use direction) are colored red. Blue arrows indicate control inputs. Example adopted from Click and Paleczny (1995).

their affiliation to a region. Not all instruction need this assignment. Instructions without this input, called *floating* instruction, can be placed anywhere between their inputs and their usages. Phi nodes are used to collect the data in case of control-flow joins. Their data inputs are aligned to the associated region. Dependencies which are introduced by memory stores and loads are handled like values. They take a data edge for the last memory operation as input. This ensures that the semantics are adhered to. Figure 2.2 shows an example of this representation.

The graph-based representation is not well suited for fast direct code generation. In his PhD thesis, Click (1995) described methods for scheduling the IR into a linear list of instructions for this purpose. Section 4.4 discusses the algorithm in detail.

Click and Paleczny implemented the nodes using a C++ class hierarchy. Instead of switch clauses, instruction-dependent behavior is implemented via *virtual functions*. This encapsulation leads to a more maintainable and extensible code.

2.2 Adaptive Optimization and On-Stack Replacement

Adaptive optimization and on-stack replacement build the core of most modern, high performing virtual machines. About one decade after the advent of these techniques in the context of virtual machines Arnold et al. (2005) presented a thorough survey on the approaches proposed for this

purpose. They divided the field into four categories, namely

1. *selective optimization*,
2. *profiling techniques for feedback-directed optimization* (FDO),
3. *feedback-directed code generation* and
4. *other FDOs*.

Selective optimization handles methods differently depending whether they are *hot*, meaning executed frequently, or not. Empirical evidence, for instance findings by Knuth (1971), suggests that it makes sense to concentrate only on the small fractions of the code that runs most of the time. Initially the code is executed using a low-overhead method, for example interpretation or a fast and simple JIT compiler. The latter is often referred to as a *baseline compiler*. If some parts of the code get *hot*, an optimizing compiler creates a high-quality version of the code. The virtual machine must supply means to replace the cheap version with the optimized one. If this can happen while the method is active it is called *on-stack replacement* (OSR). This is a non-trivial task. The concrete implementations discussed in the following sections will give more insight into this matter.

In order to know which methods are hot some sort of *profiling* is performed. *Hardware Performance Counters* are specialized facilities provided by the processor to supervise a program. *Stack Sampling* means to interrupt the normal program execution and to analyze the call stack of all threads. It is either controlled by a timer or based on some other event, for instance the number of method invocations. The quality of the result is highly dependent on the interruption frequency. An alternative is *Program Instrumentation*. Each method is associated with a counter that increments each time the method is executed. To capture hot loops, a jump to a loop header also increases the value. If the counter exceeds a predefined threshold the method is considered *hot*. This approach comes at the cost of maintaining a counter for each method. Additionally, the run-time overhead for incrementing may be enormous. Think of *setter/getter* methods, which are common in object-oriented languages, where the body often consists of only a single load/store instruction. In this case instrumentation doubles the size of the method.

The profiling data does not only invoke the optimizing compiler but also guides optimization decisions. Perhaps the most important optimization in this context is *inlining*. This transformation replaces a call instruction with the body of the callee. It increases the size of the basic block and therefore creates more opportunities for subsequent optimizations. Inlining comes not without cost though. First, the code for the inlined method is duplicated, so overeager inlining can exceed code memory limits. Second, inlining increases the amount of time spent in the compiler. Profiling information can help to select only adequate candidates for this optimization. Inlining is not the only optimization that can profit from profiling. *Multiversioning* is a transformation where different versions of the same method are created. Each version is optimized towards another goal. At run-time the best fitting version is selected. Profiling data can help to decide where multiversioning is promising. Feedback-directed optimizations not targeting code generation have also ben proposed. This category includes optimizations with regards of locality, caching and memory management improvements.

The following subsections discuss a selection of concrete systems in detail.

SELF

Chambers and Ungar (1991) first described on-stack replacement in the context of the interactive program environment for the SELF system to support source level debugging of optimized machine code. In his PhD thesis Chambers (1992) introduced *virtual states* (source states) and *physical states* (machine states). The optimized machine code, together with the call-stack and the register file, forms the physical state of the execution. For the debugger, this machine state is mapped to a virtual state of the source program. This includes translating machine stores (registers, stack-slots) back to variables, physical program counter positions to source code lines and creating one or more virtual activation records from the physical frame. This mapping information is referred to as *scope descriptions*. Chambers pointed out that the mapping from source to machine code is not a one-to-one but a many-to-many relation. In the context of inlining, several locations in the compiled code may refer to a single source code line. On the other hand, some optimizations eliminate duplicated code in which case one machine instruction is associated with many positions in the source program. The information needed for this translation is stored together with the machine code. According to Chambers this comes at the cost of high space consumption in the order of 1.5 to 5 times the size of the machine code. One solution to this problem is to page out debugging information of methods that are currently not debugged. Another suggestion he proposed is to re-execute the compiler to generate the information on demand.

Some optimizations, such as *tail call optimization* or *dead variable elimination*, can not be undone. In order to support debugging SELF does only perform optimizations where the source state can still be reconstructed. To make the translation from physical locations to variables easier lifetime splitting is not supported.

The SELF compiler already performed speculative code generation. Based on profiling data and heuristics the system assumes that an object does not change its type during execution. This allows the compiler to perform more aggressive optimizations such as inlining of virtual functions. These assumptions can get invalid during the run-time of a program. The virtual machine must react to this situation to comply with the program's semantics. The solution used by Chambers is to flush all methods that are compiled using the invalidated assumption. To avoid long compilation pauses the flushed methods are not recompiled at once but instead when needed. SELF used *dependency links* to store information about method interdependencies. If a method that is currently active is recompiled it is invalidated *lazily*. That means the return address of the activation record, that would return to the invalidated method is modified to call a special runtime procedure. This procedure prepares the stack-frame for the new version of the method. The adjustment function uses the physical-virtual translation that is also responsible for debugging.

Hölzle et al. (1992) used *interrupt points* to describe discrete positions in the machine code where the debugger could interrupt the program. In the original approach these interrupt points where placed at every source instruction boundary. It turned out to be too costly in terms of space required to store the translation information. To reduce the overhead, interrupt points are only inserted in method headers and loop back-edges. This is sufficient to ensure that an interrupt point is reached eventually. Additionally, exception throwing instructions are equipped with debug information in order to provide enough knowledge for the debugger. In between

those interrupt points the compiler can perform extensive optimizations. Hölzle et al. also used interrupt points to interact with the garbage collector. This way the compiler can ignore garbage collection in-between those points. *Customization* is another approach proposed by the authors of the SELF system. Multiple versions of a method are compiled for a particular type. This allows virtual functions to be bound statically. Related to this approach is *splitting* where multiple copies of an expression are created, each one optimized for a specific type.

The methods for debugging optimized code developed for the SELF systems soon found application in the field of adaptive optimization in other virtual machines.

Java HotSpot

One of the first popular adoptions has been the Java *HotSpot Server Compiler* developed by Paleczny et al. (2001). The Java HotSpot Virtual Machine consists of two run-time modes. At the beginning Java bytecode is executed by an interpreter. This interpreter identifies *hot methods* by counting method entries and loop back-edges. If a method hits a threshold it is selected for JIT compilation. The server compiler uses the intermediate representation developed by Click and Paleczny (1995). The edges of the IR are additionally equipped with information about the type. This includes *null*, *not-null* or specific class hierarchy glues, to name just a few. These information are either provided by analysis, for instance Class Hierarchy Analysis (CHA), or by inserting checks and splitting the control-flow into two paths. This is related to the customization and splitting approach in SELF.

Some assumptions, for example information provided by the CHA, may be invalidated by dynamic class loading. In this situation the machine code is deoptimized which means the machine stack is translated into an interpreter stack. This can be done at discrete points in the machine code, called *safepoints*, which are basically the same as *interrupt points* in SELF. For register allocation the HotSpot server compiler uses a graph-coloring approach based on Chaitin (1982) and Briggs et al. (1989).

The Java *HotSpot Client Compiler* is an alternative to the server compiler with the focus on short start-up and compilation times. For the version 6 of the Java platform Kotzmann et al. (2008) redesigned this compiler to further improve performance. Similarly to the server compiler the client compiler is invoked by the interpreter for hot methods. The redesign features two intermediate representations. The high-level IR (HIR) is a graph-based SSA representation which is based on the approach by Click and Paleczny (1995). Instructions are modeled as a C++ class hierarchy and operands are implemented as pointers to preceding definitions (use-def edge). In contrast to Click and Paleczny (1995), Kotzmann et al. use an explicit control-flow graph, whose nodes are basic blocks. Instructions are stored as a linked list in these basic blocks. No CFG edges are added from *throwing* instructions to exception handlers. This leads to a sparse control-flow graph but also means that control can be transfered from the middle of a basic block. Choi et al. (1999) introduced this concept by the name *Factored Control-flow Graph* (FCFG). The HIR is constructed in two passes over the bytecode. First, basic blocks and loop headers are created. Afterwards the basic blocks are filled by abstract interpretation of the bytecode instructions. Short, statically bound, methods are already inlined during the creation of the HIR. Similar to the server compiler (Paleczny et al., 2001) optimizations, most important inlining, are

performed on optimistic assumptions. If these assumptions are not met, deoptimization takes place and the execution continues in the interpreter.

For register allocation and code emission the HIR is translated into a low-level IR (LIR) which is close to three-operand machine code. The infrastructure is still target independent but it contains platform dependent code. The LIR reuses the CFG from the high-level representation so no new notion of basic blocks is introduced. In contrast to the HIR the LIR is not in SSA from[1] and uses explicit operands such as registers, memory addresses, stack-slots and constants. Architectural constraints, like machine instructions that require special registers, are already expressed in the LIR. Register allocation is done by an extended version of the linear-scan algorithm (Wimmer and Mössenböck, 2005). More details on the register allocator can be found in Section 4.9.

Similar to the server compiler the client version uses deoptimization to deal with invalidated assumptions. In case of class hierarchy assumptions dependencies between class and machine code method are recorded (*dependency links*). Instead of deoptimizing a method in place the instruction following the current program counter is patched to call the run-time system. Entry points to invalidated methods are also patched to direct the control to the interpreter.

Jalapeño JVM (Jikes RVM)

The Jalapeño JVM (now Jikes RVM) is a research virtual machine written purely in Java (Alpern et al., 1999). The compiler is used to optimize both, user code as well as the JVM itself. They follow a compile only strategy with two different compilers. The *baseline* compiler translates the bytecode to machine code that simply simulates the JVM operand stack. No register allocation is performed. Arnold et al. (2000) pointed out that this performs only slightly better than the interpretive approach. The *optimizing compiler* translates the bytecode into a factored CFG (Choi et al., 1999) with basic blocks and instruction lists. The linear-scan algorithm is used for register allocation. Jalapeño supports three optimization levels with increasing complexity, compile time but also code quality. The first level mainly applies parse-time optimizations such as constant, type and copy propagation. The second stage additionally performs local optimizations, for instance common subexpression elimination or array bounds check. It also performs inlining of statically known methods and guarded inlining. The last level uses SSA form to apply flow-sensitive optimizations.

The *adaptive optimization system* is used to control the evolution of machine code. Based on the estimated use/cost rate it decides which optimization level to use. Jalapeño makes heavy use of profiling data. Different subsystems are used to provide run-time information to the compiler. The information is gathered using a number of techniques including hardware performance monitors, stack sampling or edge and path profiling. The *raw data* is processed by dedicated *organizer* threads.

As soon as a hot method is detected it is queued for recompilation. To keep the memory consumption in bounds, infrequently used (*cold*) methods are flushed from the code buffer. To prevent the system from oscillating between two or more sets of optimizations previous compiler decisions are stored.

[1] This was the case in the original implementation. In later revisions SSA form destruction is done on the LIR.

Fink and Qian (2003) described the on-stack replacement strategy used by the virtual machine. Because of the compile-only approach this means to move from one compiled version of a bytecode method to another. The interface between these versions is the *JVM scope descriptor* which is analogous to the *scope descriptor* in SELF (Hölzle et al., 1992) already mentioned above. This JVM scope descriptor stores the position in the bytecode method, the content of the JVM stack and the value of local variables in a compiler independent way. Using this information Fink and Qian create a special piece of code that takes care of the transition between methods. This translation is expressed in bytecode and the compiler is responsible for translating it into machine code.

The JVM scope descriptor only can be extracted at special points of the method called *OSR Points*. These are comparable to *safepoints* in HotSpot or *interrupt points* in SELF. OSR Points are implemented as an IR instruction whose semantics are similar to a call operation. This way Fink and Qian can guarantee that all information needed for JVM scope descriptors is available.

Graal VM

Graal VM (Graal Project) is built on top of the HotSpot VM. It features a customized compilation queue and a compiler mostly written in Java. Similar to the client and the server implementation, the Graal compiler is invoked by the interpreter. In case of deoptimization of compiled code, control is transfered back to the interpreter.

The compiler used in the Graal Project uses a representation (Duboscq et al., 2013a) similar to Click and Paleczny. Instead of using Region and If nodes control-flow dependencies are handled using pairs of Begin and End nodes. Begin nodes represent the target of control transfer whereas End nodes are the source. Another difference to the approach of Click and Paleczny is the reverse direction of the CFG edges. In the approach used by Duboscq et al. an ordered list of predecessors is stored in the target Begin node.

The IR is specified in a declarative manner. It makes heavy use of Java language features. For example, annotations are used to define named edges for nodes instead of using an array with indices. The same information is used to generate reverse edges or edge iterators while the definition of nodes stays simple and maintainable.

Graal relies on speculative optimization and deoptimization. This capability is a first class member of the IR (Duboscq et al., 2013b). Similar to previous approaches, Graal needs some notion of *deoptimization points* but instead of fixing these points to the node that requires deoptimization, a more flexible approach is used. Nodes which effect the global state, such as memory writes or method invocations, are referred to as *state split* nodes. At these instructions the effect of operations executed in the current method becomes visible to the outside world. In Graal IR a special node called FrameState is used to record these state splitting operations. A FrameState node stores the method and the bytecode index. The input edges of a FrameState node are local variables and stack-slots needed for reconstructing the interpreter state.

During code emission every potential deoptimization point is associated with the last dominating state split node and its FrameState node. In case of deoptimization the interpreter restarts at the position stored by the FrameState. It may be the case that instructions have already been executed by the compiled code but because of the state splitting semantics these changes are not

visible to the interpreter so reexecution is no problem. Because FrameState nodes are floating, the compiler has many opportunities for optimization.

In principal, Graal IR models exceptions as explicit edges to the exception handling block. Due to optimistic assumptions most of these edges are replaced by guards which trigger deoptimization in the case that an exception is thrown. Deoptimization is an expensive operation and applications which use exceptions for normal control-flow would degrade by this approach. To overcome this problem frequently thrown exceptions are not replaced by deoptimizing guards.

Stadler et al. (2012) described the *Graph Caching* mechanism used by Graal VM. The ideas is to cache IR graphs of frequently compiled methods to prevent the repetitive parsing of the same bytecode method. This is profitable especially in the context of inlining where one bytecode method is compiled into many machine code methods. In the same work Stadler et al. suggest that inlining should not be done in course of bytecode parsing. Instead of this traditional approach inlining in Graal is performed by copying the parsed IR of the callee into the callers IR graph. This *late inlining* has the advantage that the parser is less complex and that on the IR level better decisions can be made due to the availability of more precise analysis.

To be competitive the Graal compiler performs a vast amount of optimizations (Stadler et al., 2013). Besides classic optimizations, such as tail duplication or loop unrolling, some transformations are specifically targeted on the Java Platform. *Intrinsification* is an approach where commonly used library methods, which are implemented using native method, are replaced by equivalent implementations that are visible to the compiler. The problem with these native method is twofold. First, Java Native Interface (JNI) (Liang, 1999) calls are expensive due to their calling convention (Grimmer et al., 2013). Second, native methods can not be used for inlining, at least not using the standard inlining framework. Intrinsification solves this problem at the cost of reimplementing and maintaining custom versions of library methods.

CACAO VM

Steiner (2007) implemented adaptive inlining and on-stack replacement in the CACAO VM (Steiner et al., 2007). As already mentioned CACAO follows a compile-only approach with a *baseline* and an *optimizing* compiler. The baseline compiler inserts countdown traps in method headers and loop back-edges. If a threshold is met, a method is recompiled using the optimizing compiler. Steiner et al. proposed three heuristics for guiding the inline decisions. When using *aggressive depth-first inlining* the compiler tries to inline the callee recursively until the size of the method exceeds a multiple of its original body. Additionally the depth of the call tree considered for inlining is limited. The *aggressive breadth-first* strategy first inlines all calls in the base method before considering the second level of the call-tree. The termination conditions are similar to the depth-first approach. The *Knapsack Heuristic* estimates the benefit to cost ratio for inlining a specific call site. If this ratio exceeds a predefined threshold the call is inlined. The algorithm starts with a predefined cost budget and continues until no more candidate can be inlined without exceeding the remaining budget.

Steiner et al. use the notion of *execution state* and *source state*, which are similar to the concepts used in SELF by Chambers and Ungar (1991). CACAO VM uses *replacement points* for switching between different compiled version of the code.

Trace-based JIT

All systems described before use a *method based* JIT compiler which work at the scope of a single method. Multiple methods may be considered in course of inlining but always on a strict hierarchical level from caller to callee.

Bala et al. (2000) pioneered an approach which does not follow this traditional model. Instead of compiling one method at a time it operates on *traces*. A trace is a stream of basic blocks with a *single entry* but potentially *multiple exits*. The novelty of this approach is that the basic block sequence may span over several method calls and returns. Because of their simple structure, traces can be optimized aggressively with simple algorithms. The information needed for generating a trace is gathered by a run-time system, for instance an interpreter. The sequence of instructions is recorded. Once a back-edge to an already recorded instruction is found, the trace is closed. Side exits of a trace usually transfer the control back to the interpreter. To avoid the overhead of switching to the interpreter a technique called *trace merging* can be applied. If a side exit is the start of another trace the control is transfered directly to the compiled code of this trace. In some cases *compensation code* has to be inserted to adjust the stack and register layout.

Gal et al. (2006) showed that this approach perfectly fits the needs of resource constrained devices. Their light-weight implementation of a trace-based JIT compiler for a JVM reached a performance comparable with full-fledged production quality systems. The traces are recorded on bytecode level using an interpreter. Gal et al. use a SSA based representation for their compiler. Because only the relevant basic blocks are considered, bytecode parsing overhead is avoided for uncommon parts. The trace-based approach is also suitable for dynamically typed languages as shown by Gal et al. (2009). The empirical study by Garret et al. (1994) suggests that the dynamic types in the hot regions of a program are relatively stable. Based on this assumption Gal et al. generate type-specialized native code for hot traces.

The design of a method based JIT and a trace-based JIT compiler is fundamentally different. Nevertheless, Inoue et al. (2011) evaluated the implementation of a trace-based JIT compiler by modifying a method-based systems and achieved promising results.

2.3 Compiler Frameworks

LLVM

Lattner and Adve (2004) introduced the LLVM (Low-Level Virtual Machine) as framework for lifelong program transformation. Although its original goals where towards dynamic compilations, it is currently more popular in the field of static, ahead-of-time compilation. LLVM introduced a *language-independent*, typed SSA representation which is not only used internally but also visible to the user. This IR has three manifestations. First, an in-memory data structure which is used by the compiler. Second, there is a space efficient binary format which can be executed using the JIT capabilities of LLVM. Additionally there is an assembler like, human readable representation. Although being typed and in SSA form the IR has a low-level, RISC-like instruction set. Lattner and Adve point out that this fact distinguishes it from repre-

sentations used in language-focused virtual machines such as the JVM. Nevertheless, there have been efforts to use LLVM for building a Java VM (Geoffray et al., 2008).

LLVM features a modular, extensible and flexible pass pipeline. A pass scheduler takes care of phase ordering and automatically reruns passes, if an analysis is no longer valid. Due to this assembler-like representation LLVM IR is forced to use the basic block and list of instructions idiom, which makes some optimizations more difficult (Click and Paleczny, 1995).

libFIRM

libFIRM is a compiler framework library written C (Lindenmaier, 2002). The representation, called FIRM (Braun et al., 2011), is based on the IR proposed by Click and Paleczny (1995). It is close to a machine architecture and does not directly support high-level language features. libFIRM features powerful optimizations such an SSA-based coloring (Hack et al., 2006) and a PBQP[2] based register allocator (Buchwald et al., 2011). These heavy-weight transformations suggest that it is designed towards static compilations.

[2]Partitioned Boolean Quadratic Programming

'Are you stealing those LCDs?' 'Yeah, but I'm doing it while my code compiles.'

CHAPTER 3

Compiler Framework

3.1 Overview

Before going into the details of the implementation, this section should give an overview of the current situation in CACAO and where the outcome of this work fits into. All following sections will discuss the new compiler framework.

3.1.1 Status Quo

This subsection summarizes the existing components in CACAO which are relevant for the new implementation.

Adaptive Optimization Framework

The existing adaptive optimization framework was developed by Steiner (2007). It consists of the following modules.

JIT Compiler The (baseline) compiler that performs the translation from bytecode to machine code. Additionally, bytecode verification is performed.

Code Repository The code repository keeps track of emitted machine code. It is also responsible for freeing unused code memory.

Method Database This module stores the properties of methods. Optional assumptions, made during compilation, are also recorded in this database.

Replacement Mechanism As the name suggests, this component performs the replacement of invalidated code.

Linker Object layout, virtual method and interface tables are determined by this module. It also triggers class initialization.

Profiler The profiler is responsible for gathering run-time information and presents the data to the compiler.

Inliner Strictly speaking, this is not an optimization framework module but a compiler pass. It performs method inlining on IR level.

Garbage Collector The exact garbage collector (Starzinger, 2011) interface is used to find live objects and for rewriting references after compaction.[1]

Architecture Layer This layer abstracts target-dependent properties needed by the framework.

In the course of adaptive optimization the compiled code undergoes state changes. These phase shifts are mandated by the run-time system, either directly by interrupting the program or indirectly using traps. The states are as follows.

Compiler Stub During class loading a *compiler stub* is created for each method. This stub is an immediate call target for methods that are not yet compiled. It triggers the compiler the first time a method is called. This *lazy compilation* does not only decrease the start-up time but also ensures that the class initialization is done in the specific order required by the JVM specification (Lindholm et al., 2013).

Baseline with Countdown Traps At the beginning all methods are compiled with countdown traps. A countdown trap is a piece of code that is placed at the method header and at every loop back-edge. It increments a counter associated with the method. If a predefined threshold is exceeded the compiler is reinvoked.

Full Instrumentation (optional) The method is recompiled with full instrumentation. To minimize the overhead, countdown traps are used to limit the iterations a method runs with full instrumentation. The current approach naively inserts a counter for each basic block. Advanced techniques, for instance edge profiling as proposed by Ball and Larus (1994, 1996), are worth considering.

Optimization When enough profiling information is available the compiler is restarted and applies aggressive optimizations, most importantly inlining. The profiling data is utilized to direct the compiler.

Sampling (not implemented) It was planned to implement a sampling profiler (Hölzle and Ungar, 1996). This profiler periodically samples the call stacks to find further optimization potential in already optimized code.

Legacy Compiler

The pass pipeline of the legacy compiler is shown in Figure 3.1. The left branch depicts the passes used for fast compilation (baseline), whereas the right paths are taken when running in optimization mode. Depending on the build-time configuration different passes are invoked, for instance SSA based optimizations are turned off by default. The tasks of the individual passes are as follows.

Parser The parser translates bytecode into the legacy intermediate representation. Therefore, it identifies basic blocks and computes local variable renaming. Additionally, a mapping from bytecode lines to basic blocks is created.

[1] Currently, the default is a conservative Boehm (1995) garbage collector.

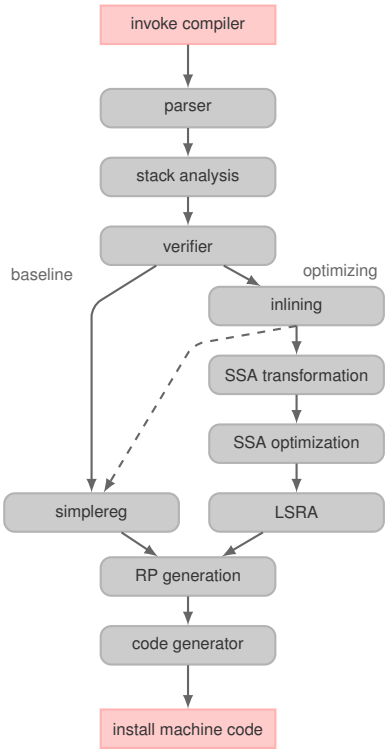


Figure 3.1: Legacy Compiler Overview

Stack Analysis The stack analysis translates the instructions from a stack-based representation into a register-based one. This is done using a simple, linear algorithm as described by Krall (1998). Additionally, the pass performs local optimizations such as subroutine elimination, constant operand optimizations, argument register pre-coloring and basic copy elimination.

Verifier Bytecode verification, which is required by the JVM specification (Lindholm et al., 2013), is done in this pass. The implementation follows the approach proposed by Coglio (2003). In addition to verification, this pass infers type information, which is used by subsequent passes.

Inlining This pass is responsible for method inlining (Steiner et al., 2007). The details can be found in the master's thesis of Steiner (2007).

SSA Transformation The SSA transformation pass translates the IR into SSA form. Therefore, local variables are split and φ-nodes introduced.

Optimization In this pass SSA based optimizations are performed such as *copy propagation* or *dead code elimination*. Transformations that are not restricted to SSA representation are *if-conversion* and *basic block reordering*. The details of these optimizations are discussed in every good textbook on compiler construction, for instance Muchnick (1997) or Aho et al. (2006).

Register Allocation The legacy compiler comes with two different register allocators. The *simplereg*, a fast and simple allocator proposed by Krall (1998), is used in the baseline

part of the compiler. For optimization, the compiler can also use a *linear scan register allocator* based on the proposal by Poletto and Sarkar (1999).

Replacement Point Generation This pass calculates the location of replacement points. These are needed for on-stack replacement and by the exact garbage collection interface.

Code generation In the last step the instructions are translated into binary code. This is done by emitting a sequence of machine instructions for each operation in the legacy IR.

3.1.2 Second-Stage Compiler

The compiler created in course of this work is designed as a drop-in replacement for the existing optimizing compiler. In contrast to the previous approach no passes from the baseline compiler are reused.[2] The input to the second-stage compiler pipeline is a method in bytecode. The result is binary machine code. The optimization branch of the legacy compiler is no longer used. The new state of affairs is depicted in Figure 3.2.

Note that the replacement point generation pass in the baseline branch is no longer required for deoptimization, if the framework follows a *lazy* replacement approach as advocated in Section 7.1. This means that the baseline compiler is reinvoked in course of on-stack replacement and the deoptimization information is generated on demand. Nevertheless, replacement points are still needed by the exact garbage collector interface, so they can not be omitted.

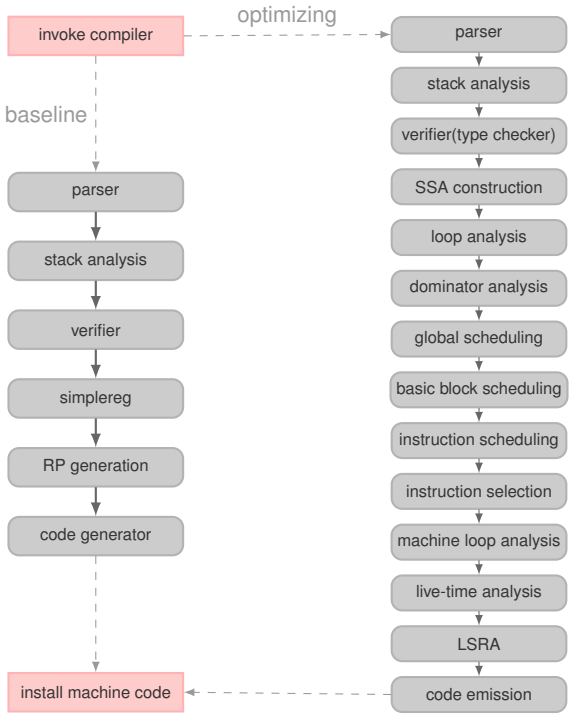
The second stage compiler is only executed for methods which have already been processed by the baseline compiler. So it is guaranteed, that the bytecode has been verified before it enters the optimization pipeline. Therefore, the reverification by the second stage compiler is superfluous. Nevertheless, the verifier is still required for inferring type information, which is needed by the SSA construction, so it can not be discarded completely.

In the following only the second-stage compiler is of interest so the *second-stage* is occasionally omitted. The new compiler consists of several components. Depending on the point of view different classification units can be distinguished. These units are depicted in Figure 3.3.

From the language perspective four different representations are used during the translation of a compilation unit. At the beginning the code is supplied as Java bytecode, which is translated into a *high-level intermediate representation* (high-level IR, or simply HIR). This representation is suitable to perform optimization and analysis passes. For transformations closer to the machine level the *low-level intermediate representation* (low-level IR, or LIR) is constructed from the high-level IR. The LIR is designed towards the requirement of register allocation and code emission. In the last step the LIR is used to emit binary machine code that can be executed by the processor.

Closely related to the language classification, the passes in the pipeline can be divided into *high-level* and *machine* passes. While high-level passes are in general working on the high-level IR whereas machine passes depend on the LIR this classification is not focused on the representation. The goal of high-level transformations is to perform optimizations and analysis to produce faster code. The machine passes, in contrast, focus on code generation and on the adherence to machine constraints. This includes register allocation and providing a code format the processor can execute.

[2] This is not entirely true. See Section 4.1 and Section 7.1.1 for more detail.



(a) Baseline Compiler (b) Second-Stage Compiler

Figure 3.2: Current Compiler Overview

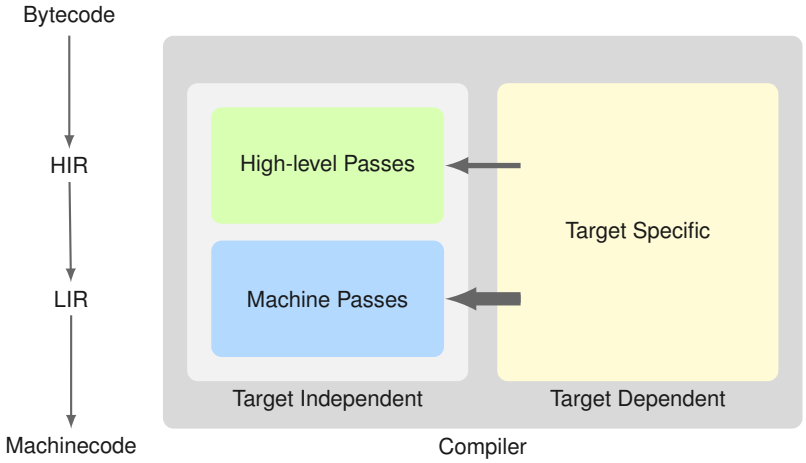


Figure 3.3: Second-Stage Compiler Overview

From an architectural point of view the implementation can be attributed to *target independent* and *target dependent* modules. The target independent part consists of the pass pipeline, the pass manager and support modules which supply static information about the overall compilation process. One example for such a supporting class is Method which stores the argument list of the currently compiled method.

The target dependent part contains all modules that are specific to one computer architecture, for instance x86_64. The responsibilities of these components include the lowering from the HIR to the LIR or providing information about the parameter passing for procedure calls. Not only the machine passes can profit from the information provided by the target implementation. High-level passes may decide which optimizations are performed based on the capabilities of the current target. Every supported architecture needs its own target dependent implementation, therefore the footprint of this part is kept as small as possible. If feasible, shared concepts should be outsourced to the target independent part. To make this possible most communication is done via abstract interfaces.

In the following sections each of these components is described in detail. For the sake of readability, data models are not contained in the flow text but collected in Appendix A.

3.2 Intermediate Representation

The compiler centers around two different representation models for the code. In the architecture independent part an SSA-based graph representation is used. The absence of a concrete schedule makes optimization and analysis passes simpler. For the architecture dependent part the graph representation is transformed into a list of basic blocks, where each block consists of a list of machine instructions. This form is more suitable for low level tasks such as register allocation and code generation. Both representations will be discussed in detail on the following pages.

3.2.1 High-level IR

The *high-level IR* is a graph-based SSA representation that follows the ideas of Click and Paleczny (Section 2.1) and Duboscq et al. (Section 2.2). A Method, which is the unit of operation for the compiler, consists of an unordered set of Instruction instances. The Instruction class is the main data structure of this representation. Instructions are also the nodes of the graph on which the high-level part of the compiler operates. The relationship between these nodes are modeled as the edges in the graph. There are three different kinds of edges, namely *control-flow edges*, *data-flow edges* and *scheduling dependency edges*.

In contrast to traditional representations, where CFG is modeled using explicit basic block data structures, this representation handles control-flow via special instructions. BeginInsts are used to join control-flow. Their functionality is similar to that of a *label* in a goto based language, like assembler. Control-flow transfers are modeled as EndInsts. An EndInst may have several successors, which are BeginInsts. Every BeginInst corresponds to exactly one EndInst. Such an instruction pair is equivalent to a traditional basic block. In contrast to BeginInsts, which are sole markers for jumps, EndInsts have a semantic function. The different semantics are implemented in subclasses of EndInst. For instance, an IFInst selects the successor depending

```
1 static long fact(long n) {
2     long res = 1;
3     while (1 < n) {
4         res *= n--;
5     }
6     return res;
7 }
```

Listing 3.1: Factorial

on its input. A RETURNInst transfers the control to the caller of the current method. These *control-flow edges* from EndInst to BeginInst form the control-flow graph of the method.

An instruction can take an arbitrary number of inputs and produces a single output value. The instruction itself is the only representative of the value it produces. Consequently, there are no variables or registers. Each value is defined exactly once, by the instruction that computes it. This is a central concept of the SSA representation. Therefore the inputs can be modeled as simple pointers to the defining instruction of a value. These pointer are called *data-flow edges* (use-def edges) and form the data-flow graph of the method. PHIInsts (φ node) join values from different control-flow paths. A PHIInst always corresponds to exactly one BeginInst. *Scheduling dependency edges* express dependencies of this kind. They are similar to data-flow edges but do not involve data transfers. They simply define an execution order between two instructions. For φ-nodes an even stronger relation is defined. PHINodes are *fixed* to a particular BeginInst. That means that it has to be scheduled somewhere between the BeginInst and the corresponding EndInst. Another category of instructions that require special attention are instructions with *side-effects*. That means, they interact with the *global state*, like for example method calls or memory accessing instruction such as PUTSTATICInst or GETSTATICInst. The interdependencies of side-effect instructions is also modeled via schedule dependency edges. Figure 3.4 illustrates the HIR graph of the Java function in Listing 3.1.

In Java exceptions are omnipresent. Potential exception-throwing instructions (PEI) have explicit edges to the exception handler block (EXH). The result is a dense CFG with reduced scheduling freedom. Once deoptimization is available, it is planned to replace these exceptional edges with guards and return to the baseline compiler if an exception is thrown.

Type information for an instruction is stored in the type attribute. This is in contrast to the baseline IR, where there is a typed version of each instruction. The attribute approach keeps the total number of instructions low, but on the other hand lowering gets more difficult. In addition to the types supported by the JVM bytecode, special values, like the global state, have a dedicated type.

The instructions are implemented as a C++ class hierarchy. Instruction is the base class for all other instructions. The date-flow edges are implemented as an ordered list of operands. Each entry of this list is a pointer to the instruction that defines the value. Scheduling dependencies are also modeled as a list of pointers. For convenience the reverse direction for data-flow and scheduling dependency edges is maintained. These reverse edges can not be modified directly.

For control-flow edges only BeginInst and EndInst are relevant. An EndInst maintains an ordered list of *successors*, which defines the CFG. Again, the BeginInst stores the reverse

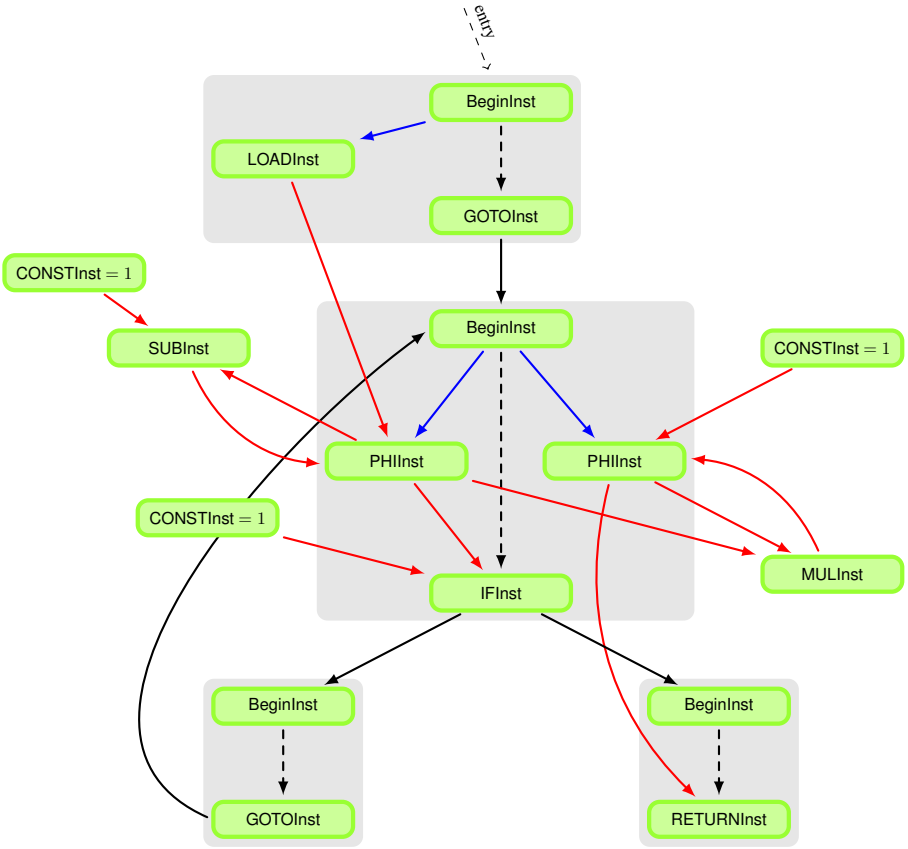


Figure 3.4: High-level IR Graph for Listing 3.1. — Solid black arrows indicate CFG edges from predecessor to successor. Associated BeginInst-EndInst pairs are marked via dashed black lines. Data edges are depicted as solid red arrows pointing into the def-use direction. Solid blue lines are scheduling dependencies.

counterpart as a list of *predecessors*. When modifying control-flow edges special care must be taken because the operands of a PHIInst are aligned to the predecessors of BeginInst. In case the ith predecessor is deleted, the ith operand of all PHIInst must be removed as well. The framework provides limited support for such modifications but it is the pass developers duty to ensure that the graph is valid after a transformation.

Every instruction class can be identified by an unique opcode. While this is sufficient to distinguish instructions, some data is only accessible through a specialized subclasses interface and not through the Instruction superclass. Important examples are the successor and predecessor lists in EndInst and BeginInst. To get access to the subclass interface the framework needs support for *dynamic casting*. The run-time type information (RTTI) mechanism provided by C++ has performance costs and should be avoided if possible (Meyers, 2005, Item 27). To circumvent this, the Instruction class contains dedicated *virtual casting functions* of the form

to_ExampleInst() for each instruction. These methods return a pointer to the special subclass or NULL if the cast is not possible. The following code snippet illustrates the usage of the casting methods:

```
1   Instruction *I = ...
2   BeginInst *BI = I->to_BeginInst();
3   ... = BI->get_predecessor(0); // only available in BeginInst
```

This approach comes not without cost. First, the virtual function pointer table for the Instruction class needs to store an entry for each subclass. Additionally, concrete instructions need to overwrite the appropriate to_XXX method. The extensibility also suffers from this approach. If a new instruction is added, a new virtual casting function needs to be added as well. This inconvenience is tolerated because new instruction are not added on a daily basis.

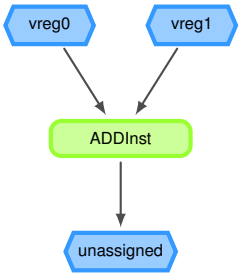
Not all relations between instructions are expressed by the subclass relations. C++ supports multi-inheritance but this approach leads to degenerated hierarchy graphs with *inheritance diamonds* (Meyers, 2005, Item 40). To avoid this problem, *properties* of an instruction are modeled via virtual Boolean member functions. For instance the method is_commutative() returns true, if the operands of an instruction are interchangeable.

3.2.2 Low-level IR

It is the goal of the high-level representation to make the implementation of optimizations as simple as possible. In contrast, the *low-level IR* is designed to support target dependent tasks like register allocation and machine code emission. It follows the classic basic block and instruction list approach.

The atomic entities are MachineInstructions. Ideally they represent exactly one instruction supported by the processor. MachineInstructions are organized in an ordered list which forms a MachineBasicBlock. The blocks are already ordered according to some block scheduling. In additional to the list of instructions, a basic block maintains other meta information to support succeeding passes. For example, a list of predecessors, which is used by the lifetime analysis pass. The LIR uses explicit operands for data transfer. Available are Constants, VirtualRegisters, MachineRegisters, VirtualStackSlots and MachineStackSlots. Virtual stack-slots are similar to virtual registers. They do not require a specific stack-slot index and can be allocated to any free slot. Note that the CFG of the LIR is more detailed than the CFG of the original HIR graph. First, some multi-target HIR instructions, such as the LOOKUPSWITCHInst, have no single equivalent native machine instruction. They are simulated by multiple conditional and unconditional jumps. Second, register allocation and SSA deconstruction introduce basic blocks to place resolution code.

Every basic block starts with a MachineLabelInst. It will never emit any code but is used to support machine dependent tasks like the code generator, which identifies jump targets by MachineLabelInst. The last instruction of each basic block is a control-flow transfer instruction like jump, conditional jump or return. Fall through semantics are modeled explicitly, meaning that, for instance, a conditional jump has two successors. One for the case the condition is met and one where it is not the case. Sometimes this violates the idea that every MachineInstruction represents exactly one hardware instruction. A conditional jump can emit two instruction. One



(a) HIR

```
... -> vreg0
... -> vreg1
mov vreg1 -> vreg2
add vreg0, vreg2 -> vreg2
```

(b) LIR

Figure 3.5: Two-Address Instruction Handling

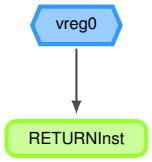
conditional jump for the *then*-path and an unconditional jump for the *else*-path. In general the *then*-block is scheduled after the block with the conditional jump. Therefore the artificial jump instruction is emitted rarely.

Because of its simple representation, an SSA-based low-level IR would be preferable. On the other hand it should model hardware instructions as precisely as possible. Unfortunately, these two goals are contradicting. One problem are two-address instructions. Two-address instructions are instructions, where one source operand is also the target operand. Another problem are instructions with fixed input or output operands. For example the return value of a function is often passed via a special register. These instructions are in conflict to the SSA properties where each variable is defined only once. During the design phase it turned out that sticking to the real instruction semantics and sacrificing some of the SSA properties led to a smaller and more intuitive implementation.

To work around the two-address instruction issue, a copy of the source operand is created. Predefined operands are handled alike. Moves are inserted right before or after the instruction. Examples for both cases are depicted in Figure 3.5 and Figure 3.6. It turns out that this approach preserves enough properties to permit the use of algorithms, originally designed for SSA representation. The properties are the *dominance property*, which means that each use of a value is dominated by a definition, and a relaxed form of the *single definition* property (Rastello, 2013). Although it is not guaranteed that each variable is defined only once due to the problems described above, each usage is reached by exactly one definition. This property will be called *single reaching definition*. To retain the properties, φ-functions are inherited from the high-level IR. MachinePhiInsts are not part of the regular instruction list but stored independently for each basic block. Conceptually, they are in parallel and located at the beginning of a block. MachineLabelInsts serve as placeholder for this position.

Creating many copies of a value poses a challenge for the register allocator. To get reasonably good results even without costly optimizations like coalescing, *register hints* (Wimmer and Mössenböck, 2005) guide the allocator into the right direction.

As already mentioned, basic blocks and instructions in the LIR are scheduled, so all instructions are in a *linear order*. The lifetime analysis and the linear scan algorithm depend on such an ordering, more precisely on the *ordering relation* of two instructions. By using an index this



(a) HIR

```
... -> vreg0
mov vreg0 -> %rax
ret
```

(b) LIR

Figure 3.6: Fixed-Operand Instruction Handling

question can be answered in constant time. One issue with the naive approach is that instruction insertion invalidates the indices. The problem of supporting both, a constant time ordering query as well as insertion and deletion of values, is known by the term *list maintenance problem* (Dietz and Sleator, 1987; Bender et al., 2002). By using a list maintenance algorithm the register allocator can insert spill instructions and new lifetime intervals without invalidating the existing lifetime analysis.

3.3 Target Implementation

Most of the target dependent code is hidden behind the Backend interface.[3] The most important task is the lowering of HIR Instructions into LIR MachineInstructions. Therefore the LoweringVisitor class is specialized by the target implementation. This visitor is used during instruction selection (Section 4.7).

The Backend provides an interface to create jumps and moves, namely create_Jump() and create_Move(). These functions are needed by machine passes, like the register allocator, to create stack stores and loads or to insert resolution code.

The target implementation also contains a model for the physical register file. Registers are implemented by subclassing MachineRegister. This way the target independent part can use them without knowing the details of the implementation. For register allocation the Backend provides a method to poll the set of physical register candidates for a given type. It is possible to define overlapping register classes. This is useful for architectures that use register pairs to hold values that do not fit into a single register. This information is incorporated into the calculation of free and occupied resources.

The target independent part also contains the definition of the MachineInstructions. As already mentioned above, they should model the real instructions supported by the processor. MachineInstructions do not only carry information but also store code to emit binary machine code.

In the course of this work a prototype implementation for the x86_64 architecture has been implemented. Extending this to other processors is left for future work. Section 7.1.1 discusses a roadmap for this endeavor.

[3] Backend is an unfortunate name because it is often associated with code generation. In this implementation it is used to denote the interface between target independent and target dependent code.

3.4 Pass Pipeline

The pass pipeline is the backbone of the compiler. It executes the compiler passes and manages the communication between them. The general idea is the following: An optimization or analysis pass has a set of inputs, for instance the control-flow graph, the data-flow graph, or analysis information like the loop or dominator tree. Additionally, a pass usually produces a result. An analysis *gathers information* about the program whereas an optimization or transformation *changes* the program.

The object-oriented paradigm suggests the encapsulation of data and functionality, therefore the pass itself is not only responsible for generating the result but also acts as a data store for the information. The LoopPass, for example, does not only calculate the loop tree but also provides an interface for other passes to access the information. Pass objects are used to transfer the results from one pass to another. Transformation passes not only access the information provided by other passes but also change it. For example constant propagation changes instructions in the data-flow graph. This modification invalidates all passes that rely on changed data. The PassManager is responsible for keeping the information up to date. This approach makes the framework easily extensible. Adding a new pass means simply to subclass Pass. No global data structures need to be altered and all other passes stay the same.

The pass manager used in LLVM (Lattner and Adve, 2004) follows a similar approach and influenced this implementation. The core classes of the pipeline are Pass, PassUsage and PassManager. The following subsections will discuss them in detail.

3.4.1 The Pass Class

Each compiler pass is a subclass of Pass. The entry point is the *pure virtual* function run(). The HIR graph can be accessed through the Method object. It contains an unordered list of Instructions, an unordered list of BeginInsts, a pointer to the entry BeginInst and meta information about the method such as the method name, the signature or the class name.

The current Method object is stored in the JITData instance, which is the only parameter of run(). JITData contains data that is persistent throughout the compilation. Additionally to Method, it provides access to a Backend instance, which can be used to retrieve information about the machine architecture.

As already indicated above, a Pass object also carries information for other passes. A previous pass can be accessed via the get_Pass<PassName>() and get_Pass_if_available<PassName>() methods in Pass. The former requires that PassName has been executed and is up to date.

The second method, get_Pass_if_available<PassName>(), returns pointer to a PassName object if it is available and up to date. This is useful for passes, which can be scheduled at different stages of the pipeline. For example a pass does not rely on instruction scheduling information but in case it is available it can improve its performance. Because this method implies no pass scheduling dependencies, the result of the pass must not rely on the information returned by this method. The result of a pass must not be invalidated if the information, retrieved using get_Pass_if_available<PassName>(), changes.

To allow other passes to access the computed results, a Pass should define a public interface. Multiple inheritance can be used to separate the presentation and the generation of the result. For an example see LoopPass and LoopTree.

During the compilation of one translation unit only one instance for each Pass subclass exists. If a pass is executed more than once, the run() method is also called on the same object. Therefore the constructor can not be used to initialize or reset data structures. To circumvent this problem the virtual methods initialize() and finalize() are executed before and after run().

For debugging purposes, Pass provides a virtual function verify(). As the name suggests it can be used to verify the result of a transformation. If the method returns *false* the compiler is stopped and the virtual machine terminates. The pass manager calls this method only when running in debug mode.

The pass manager does not need to know the individual interfaces of all passes. In order to distinguish passes, an identifier is used which is unique for each subclass of Pass. A pointer to the virtual function table (*vtbl*) would meet these requirements. Unfortunately, in C++ this pointer can not be accessed in a portable way. To work around this issue, each subclass of Pass requires a *static* member variable named ID. Note that value and type of this variable does not matter, only the address is important. This approach is not optimal because 1.) it wastes heap space for the variable and 2.) it can be spoofed by making ID a reference that points to the address of another ID. In practice both issues are acceptable. The first can be minimized by using a variable with a small memory footprint, for instance char. The second issue is simply ignored. To allow the pass manager to initialize the pass object, every Pass subclass must contain a *public* constructor without arguments. See the souce code of ExamplePass in the appendix for an example. Both requirements described above, the static ID field and the constructor, are checked by the C++ compiler at compile-time using *templates*.

A Pass is registered to the PassManager by creating a *static object* of the PassRegistry class. The *constructor* inserts the pass into the list of known passes. The initialization of static objects with non-trivial constructors is problematic because the C++ standard (ISO, 2012b) does not specify the initialization order.[4] Because the registration system does not rely on any specific order this is not a problem as long as the static constructors are executed eventually which is guaranteed (ISO, 2012b, §3.6.2, Item 4).

3.4.2 The PassUsage Class

The order in which the passes are executed is not explicit, but defined implicitly by the dependencies between passes. For this purpose a Pass can overwrite the virtual member function get_PassUsage(). Note that this function should have a static behavior, meaning it always returns the same result, independent of its current state. Via the PassUsage object a Pass p can define two kinds of *preconditions*. The add_requires<PassName>() method asserts that PassName must be executed before p and that the result of PassName must be up to date. This requirement is needed if p wants to access the result via get_Pass<PassName>(). This is also called a *strong* requirement.

[4] In fact, finding a correct order is unsolvable (Meyers, 2005, Item 4).

In contrast, add_schedule_after<PassName>() only demands that p is executed at some point after PassName. The result of PassName is not required to be valid. Therefore, p can not access its data with the get_Pass<PassName>() method. It can try to use get_Pass-_if_available<PassName>() but there are no guarantees that it will succeed.

These two preconditions are sufficient for mandatory transformations that are needed for generating code. Optional passes need to be injected into the pipeline at a specific position. For that task the methods add_run_before<PassName>() and add_schedule_before<Pass-Name>() are available. Basically they are the inverse of add_requires<PassName>() and add_schedule_after<PassName>(). The first is used to place Pass p before PassName and to guarantee that it is up to date. The MachineInstructionPrinterPass, for example, uses this method to insert itself before register allocation and also before code generation. Because one of its dependencies, the machine instruction schedule, is modified by the register allocation, the printer pass is scheduled twice. The second method, add_schedule_before<PassName>(), instructs the pass manager to schedule p somewhere before PassName. There is no guarantee that p is still valid when PassName runs.

Postconditions for a pass p are also specified using the PassUsage object. The member method add_destroys<PassName>() informs the PassManager that the result PassName is no longer valid after the execution of p. This information is recursively propagated to all passes strongly depending on PassName. It does not automatically reschedule these passes, though. Only if the result of an invalidated pass is needed later on, it is rescheduled. The other post-condition is add_modifies<PassName>(). It can be used to inform the PassManager that the information associated with a pass has been changed. This assertion invalidates all passes that strongly depend on PassName but not PassName itself. For instance register allocation modifies the LIR.

All these dependencies are not sole data requirements but are also assumptions a pass makes about the state of the compiler pipeline. For example, depending on the ListSchedulingPass asserts that all instructions are assigned to a basic block and are in a linear order. A pass which depends on the register allocator does not expect to encounter virtual machine operands in the LIR.

Figure 3.7 illustrates the dependency graph for the standard passes. As already mentioned above, the HIR graph is stored in Method and not, like other information, in a Pass. The reason therefore is that this graph is required by all passes. Nevertheless, passes need information about the modification of the HIR graph. This task is handled by two meta passes, namely CFGMetaPass and InstructionMetaPass. Both have an empty run() method and are solely responsible for propagating changes to other passes. The first pass captures CFG modifications like basic block additions, removals or modifications. The second meta pass informs the pipeline about new, deleted or modified instructions. Note that changes to the CFG always imply changes to instructions because of the BeginInst-EndInst representation of basic blocks.

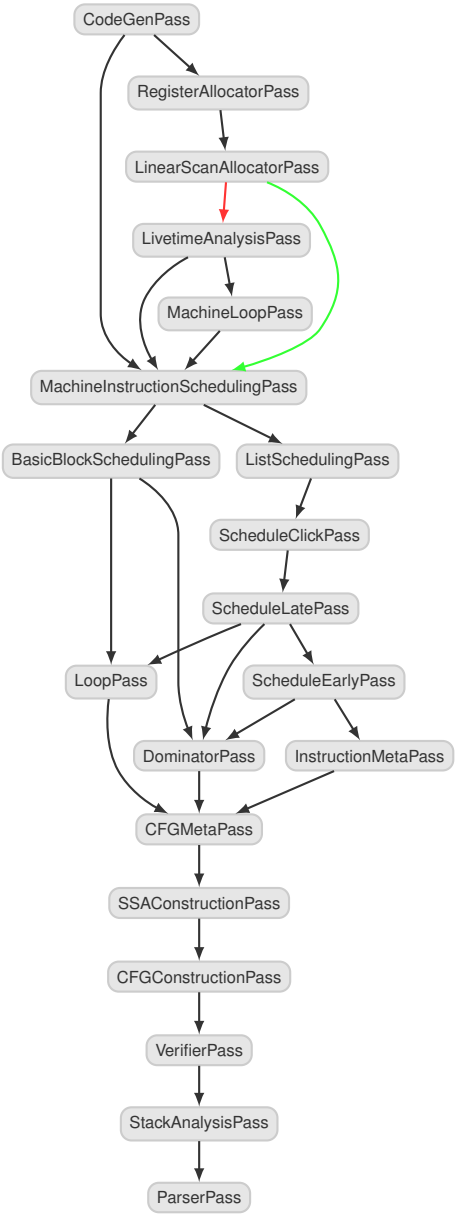


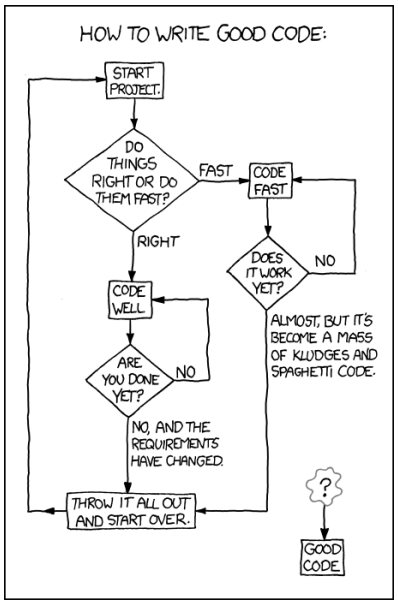
Figure 3.7: Pass Interdependencies — Black edges are *strong* dependencies. *Modifications* are illustrated with green arrows. A red edge denotes a *destroys* postcondition. RegisterAllocatorPass is a meta-pass which simplifies the implementation of different allocation algorithms.

3.4.3 The PassManager Class

The PassManager contains the implementation of all functionalities provided by the pass pipeline. It is responsible for

- constructing and storing the Pass objects,
- scheduling the passes according to their dependencies,
- initializing, running and finalizing the passes, and
- making the result of a pass available for subsequent passes.

Scheduling is done before the passes are executed, hence pass dependencies are static. It uses a list scheduling (Aho et al., 2006) based algorithm. The standard approach is extended with the capability to track the modification of passes and reschedule them if possible. There are two possible sources for non-termination in this algorithm. The first is circular dependencies. This issue occurs when two passes depend on each other. The other problem comes with the invalidation and rescheduling of passes, more precisely: if a pass has two dependencies which invalidate each other. Both issues indicate a misconception about the requirements of a pass. The pass developers are responsible for avoiding such errors. Due to performance considerations the PassManager does not detect these problems.



You can either hang out in the Android Loop or the HURD loop.

CHAPTER 4

Compiler Passes

4.1 SSA-Graph Construction

The SSA construction is based on the algorithm proposed by Braun et al. (2013). The input representation is the IR used by the baseline compiler. It is organized in a control-flow graph of basic blocks which consists of a list of instructions. The instructions are similar to the JVM instructions with the difference that stack access is replaced by variables. The details of this translation scheme are described by Krall (1998). The JVM specification (Lindholm et al., 2013) requires that every class file is verified at link-time. This is already done during the construction of the baseline IR.

The advantage over Cytron et al. (1991)'s algorithm is that the approach by Braun et al. does not require any additional analysis information to compute minimal and pruned SSA form.

For every basic block in dominance order the algorithm iterates over all instructions in execution order. For each *write*-access to a variable v the writing-instruction is recorded as the *current definition* of v. If an instruction *reads* from variable v, the current definition of v is stored as an operand of the instruction. A basic block is *filled* when all instructions in the block are handled. In literature this approach is called *local value numbering* (Muchnick, 1997).

If the definition of a variable v is not found in the current block, the algorithm searches all predecessors in a recursive fashion. To join the definitions from the predecessors a new φ-node is created and recorded as the current definition of v. More precisely, this is done *before* the recursive lookup is initiated. This way the algorithm terminates in case loops are present. The operands of the φ-node are added after the lookup. Literature calls this procedure *global value numbering* (Muchnick, 1997). Global value numbering introduces *trivial* φ-nodes which only depend on themself and one other value. These nodes are removed in an extra recursive traversal over newly created φ-instructions.

```
1  Procedure write_variable(variable, block, value)
2      currentDef[variable][block] ← value
3  Procedure read_variable(variable, block)
4      if currentDef[variable] contains block then
           // local value numbering
5          return currentDef[variable][block]
6      end
       // global value numbering
7      return read_variable_recursive(variable, block)
```

Algorithm 4.1: Local Value Numbering

```
1  Procedure read_variable_recursive(variable, block)
2      if block not in sealedBlocks then
           // Incomplete CFG
3          val ← new Phi(block)
4          incompletePhis[block][variable] ← val
5      else if |block.preds| = 1 then
           // Optimize the common case of one predecessor:  No phi needed
6          val ← read_variable(variable, block.preds[0])
7      else
           // Break potential cycles with operandless phi
8          val ← new Phi(block)
9          write_variable(variable, block, val)
10         val ← add_phi_operands(variable, val)
11     end
12     write_variable(variable, block, val)
13     return val
14 Procedure add_phi_operands(variable, phi)
       // Determine operands from predecessors
15     foreach pred in phi.block.preds do
16         phi.appendOperand(read_variable(variable, pred))
17     end
18     return tryRemoveTrivialPhi(phi)
```

Algorithm 4.2: Global Value Numbering

Local and global value numbering are sufficient for CFGs without loops. Due to the dominance traversal of basic blocks, all predecessors of the current block b are filled before b is handled. The presence of loops breaks this property. To tackle this problem the notion of a *sealed* block is introduced. A block is sealed if all its predecessors are filled. If an unsealed block does not provide a definition for a variable v, an empty φ-node is inserted and recorded as the current definition of v. Later when a block gets sealed, the operands of empty φ-instructions are fixed and trivial φ-nodes are removed.

The algorithm described above does not create minimal SSA-form for *irreducible control-flow*. That means that there are redundant φ functions. Braun et al. propose a solution to this issue. The implementation developed in course of this work does not yet incorporate this enhancement. Structured programming languages, such as Java, can never produce irreducible

```
1  Procedure try_remove_trivial_phi(phi)
2      same ← None
3      foreach op in phi.operands do
4          if op = same ∨ op = phi then
               // Unique value or self-reference
5              continue
6          end
7          if same = None then
               // The phi merges at least two values:  not trivial
8              return phi
9          end
10         same ← op
11     end
12     if same = None then
           // The phi is unreachable or in the start block
13         same ← new Undef()
14     end
       // Remember all users except the phi itself
15     users ← phi.users.remove(phi)
       // Reroute all uses of phi to same and remove phi
16     phi.replaceBy(same)
       // Try to recursively remove all phi users, which might have become
          trivial
17     foreach use in users do
18         if use is a Phi then
19             try_remove_trivial_phi(use)
20         end
21     end
22     return same
```

Algorithm 4.3: Remove Trivial φ-Function

```
1  Procedure seal_block(block)
2      foreach variable in incompletePhis[block] do
3          add_phi_operands(variable, incompletePhis[block][variable])
4      end
5      sealedBlocks.add(block)
```

Algorithm 4.4: Block Sealing

CFGs (Kosaraju, 1973), although, JVM bytecode can still produce irreducible control-flow it is not the common case.

For self-containment of this presentation, the pseudo codes from the original literature (Braun et al., 2013) are presented in Algorithm 4.1-4.4. In the following paragraph idiosyncrasies with regards to this work are discussed.

As mentioned above, the input to the SSA construction pass is the baseline compiler IR. So the pass does not only establish SSA-form but also creates the HIR instruction graph from scratch. Because the concept of variables is absent, there is no difference between program and temporal variables. A mapping from HIR instructions to variables can be established by

creating an artificial instruction with data edges to all variables. This is needed, for instance, for deoptimization points where the machine state is translated to the source state.

The SSA construction pass guarantees that the entry block has no predecessors. A new block is created, if this is not already the case in the input representation. The entry block is used for initialization code, like loading method parameters.

Because the HIR does not maintain an instruction schedule, instructions with side-effects must be handled with care. Therefore the SSA construction uses a virtual *global state* variable to capture changes which are visible to the outside world. All side-effect instructions get an input dependency to the current definition of this variable and do also set the current definition. Note that this is a conservative approximation of the real state of affairs. Handling *read-after-write* (RAW or flow/true-dependency), *write-after-read* (WAR or anti-dependency), *read-after-read* (RAR or input dependency) and *write-after-write* (WAW or output-dependency) independently would remove unnecessary restrictions. Additionally, side-effect instructions are currently fixed to a basic block. If dependencies are handled more accurately, this constraint could be relaxed.

4.2 Loop Analysis

This analysis is used to identify loops in the CFG. A *loop* is defined by an edge from the loop exit to the loop header. The result of the loop analysis is a *loop tree*. This tree reassembles the nesting of loops.

The algorithm used in this work is based on the method proposed by Tarjan (1974a). Therefore the BeginInsts (blocks) of the current method are traversed using *Depth First Search* (DFS). Every edge from v to w in this search tree is either

- a *forward edge* if w is a descendant of v , or
- a *cycle edge* if v is a descendant of w, or else
- a *cross edge*.

Note that Aho et al. calls forward edges *advancing edges* and cycle edges *retreating edges*. The *descendant* relation can be calculated in constant time by storing the preorder number and the number of descendants (Tarjan, 1974b).

Afterwards the vertices of the DFS tree are visited in reverse preorder. For each target of a cycle edge a Loop data structure is allocated. All blocks on a path from the header to the exit are part of the loop. If these blocks of loop l are a superset of another loop l', then l' is an inner loop of l. This instance of the *disjoint set* problem can be solved efficiently using *Union Find* method also proposed by Tarjan (1975). Because the vertices are visited in reverse order, inner loops are always processed before their parent loops as long as they do not share the loop header. To handle this corner case Vick (1994) refined the original algorithm by ordering cycle edges which share the header according to the preorder number of their exits, from low to high. Another issue that is handled this way, is the case where two loops share a header but their blocks both are a subset nor a superset of each other. The algorithm arbitrary selects the loop, where the exit has the lower preorder number, as the inner loop. This approximation allows the handling of *irreducible loops*, which are loops with side entries.

4.3 Dominator Analysis

Goal of the dominator analysis is calculating the *dominance* (dom) relation. Let d, n be vertices of the CFG graph then $(d, n) \in dom$ if every path from the entry of the graph to n contains d (Aho et al., 2006). A space efficient representation of this information is a tree where vertices are the same as in the CFG graph and the root of this tree is the entry node. This tree is called *dominator tree*. Due to the properties of the dom relation the dominator tree always exists and is unique (Aho et al., 2006). This implies the existence of an unique direct ancestor, the parent, for each vertex except for the root. In the context of the dominator tree this is called the *immediate dominator* or the $idom$ relation (or more accurately function). If $w = idom(v)$ then w is the vertex closest to v which occurs on all paths from the entry vertex to v. In other words $idom(v)$ is the $w \in dom(v)$ with minimal distance from w to v. Storing the $idom$ for each vertex requires only $|V|$ space, where $|V|$ is the number of vertices. The dom relation can be restored by calculating the *transitive closure* of the $idom$ function. This makes the dominator tree an effective data structure for storing dominance information.

Different algorithms have been proposed for calculating the dominance relation. Probably the most popular is the iterative *data-flow analysis* as described in Aho et al. (2006). The method implemented in course of this thesis was proposed by Lengauer and Tarjan (1979). It directly calculates the dominator tree which makes it more suitable than the data-flow approach, which goes the detour over the dom relation. Similar to the loop analysis the algorithm is based on depth-first traversal and preorder numbering. This DFS tree is transformed in a dominance preserving way in order to generate the dominator tree. Lengauer and Tarjan proposed two variants of their algorithm which differ only in the details of this transformation. Currently the *simple* version is implemented. The *sophisticated* version is left for future work.

4.4 Global Scheduling

The term *global scheduling* is used ambiguously in the compiler construction community. For example Aho et al. (2006) use it to describe methods that may move instructions from one basic block to another. In the context of this work a global schedule is the mapping of instructions to basic blocks. The difference to Aho et al. is that inside basic blocks no ordering is established. This mapping is necessary, because the high-level IR does not maintain such an assignment for most instructions. Finding this mapping is not trivial, because there is not only one candidate block for each instruction but possibly dozens. Click (1995, Chapter 6) defines the range of candidate blocks as follows: First, all floating instructions are scheduled as early as possible. That is the first basic block where the instruction is *dominated* by all its inputs.[1] This approach schedules instructions out of branches which leads to a lot of speculative code. The latest position can be found by calculating the lowest common ancestor of all users of a node in the dominator tree. Note that the block in the early schedule always dominates the block of the late schedule. As a result the instruction can be placed anywhere along the path between these two blocks on the dominator tree. To determine the final position Click uses a simple heuristic. Schedule an

[1] Per definition a node dominates itself.

```
Global: A list of BeginInst scheduled
1  Procedure schedule_loop(Loop loop)
2      blocks ← blocks associated with loop
3      subloops ← subloops of loop
4      while ¬empty(blocks) ∨ ¬empty(blocks) do                  // loop not finished
5          while ∃ b ∈ blocks | idom(b) ∈ scheduled do        // a block can be scheduled
6              blocks.remove(b)
7              scheduled.append(b)
8          end
9          while ∃ l ∈ subloops | idom(header(l)) ∈ scheduled do      // schedule subloop
10             subloops.remove(l)
11             schedule_loop(l)
12         end
13     end
```

Algorithm 4.5: Basic Block Scheduling

instruction as late as possible, but outside of loops. This way instructions are only executed if really needed but still not recalculated inside a loop body.

4.5 Basic Block Scheduling

Basic block scheduling is the process of bringing the vertices of the control-flow graph, the BeginInsts, into linear order. The framework could work with an arbitrary arrangement but later passes assume a special basic block ordering. Basic blocks are required to be in *dominance order*, meaning all blocks dominating a basic block b must be scheduled before b. This ensures that values are defined before they are used, when iterating the list of basic blocks. The operands of φ-instructions are excluded by this rule. Another requirement is that blocks of a loop are scheduled contiguously, so that two blocks of the same loop are not separated by a non-loop block. This is done by a simple algorithm that recursively schedules blocks from outer to inner loops. The pseudo code is depicted in Algorithm 4.5.

4.6 Instruction Scheduling

In this context instruction scheduling means bringing all instruction in one basic block into sequence. This is done using a classic list scheduling algorithm as described by Aho et al. (2006). Due to the SSA form loop back-edges need no special handling. Besides data dependencies, scheduling dependencies are also taken into account.

4.7 Machine Instruction Selection

Instruction selection is done by simply translating one Instruction into a series of MachineInstructions. As already mentioned above, values are modeled explicitly by MachineOperands. First, a MachineBasicBlock is created for each BeginInst. This mapping is used to set the

Input : A BasicBlockSchedule BBS
Output: A MachineInstructionSchedule

```
1  MIS ← new MachineInstructionSchedule
2  map_bb ← new map from BeginInst →MachineBasicBlock
3  map_inst ← new map from Instruction →MachineOperand
4  foreach BeginInst BI ∈ BBS do                          // create machine basic blocks
5  |  MBB ← new MachineBasicBlock
6  |  MIS.insert(MBB)
7  |  map_bb[BI] ← MBB
8  end
9  foreach BeginInst BI ∈ BBS do                          // lower instructions
10 |  MBB ← map_bb[BI]
11 |  visitor ← new LoweringVisitor(BI, MBB, map_bb, map_inst)
12 |  foreach Instruction I scheduled to BI do           // lower instructions
13 |  |  I.accept(visitor)
14 |  end
15 |  foreach BeginInst pred ∈ predecessors(BI) do       // fix predecessors
16 |  |  MBB.insert_predecessor(pred)
17 |  end
18 end
19 foreach MachineBasicBlock MBB ∈ MIS do                // fix control-flow and phis
20 |  foreach MachinePhiInst phi ∈ MBB do                // fix phi instructions
21 |  |  update_phi_operands(phi)
22 |  end
23 |  foreach jump ∈ MBB where jump.is_jump() = true do  // split basic blocks
24 |  |  MBB' ← new MachineBasicBlock
25 |  |  MBB.insert_after(MBB')
26 |  |  move_instructions(jump,MBB.last(),MBB)
27 |  |  break
28 |  end
29 end
30 return MIS
```

Algorithm 4.6: Instruction Selection

control-flow targets for jump instructions. The instruction selection is done by iterating over all instructions in dominance order. Due to our special basic block scheduling, this can be achieved by a simple iteration. The φ-nodes have to be handled slightly different than other instructions, because their input operands are not known in case of loop back-edges. This is tackled similarly as it is done during HIR construction (Section 4.1). Each PHIInst is translated into a MachinePhiInst without operands. These operands are adjusted in a subsequent traversal. Some HIR control-flow transfer instructions, for example LOOKUPSWITCHInst, can not be represented by a single LIR jump instruction. In these cases new machine basic blocks are created to guarantee the invariant, that every block ends with exactly one transfer instruction. The procedure, which is depicted in Algorithm 4.6, is implemented in the MachineInstruction-SchedulingPass.

4.7.1 Lowering

Expressing the semantics of a single HIR instruction by means of instructions supported by the processor is called *lowering*. Floating point arithmetics are replaced by calls to subroutines if the target does not support these operations natively. Lowering is done using the *Visitor Pattern* (Gamma et al., 1995). This way each target can implement a transformation for each Instruction by overwriting the corresponding visit() method from the LoweringVisitor. The target implementation is responsible for inserting the instructions in the correct order. The lowering visitor is not required to take care of basic block splits due to multiple end instructions. As mentioned above, this is done in a separated phase.

4.8 Lifetime Analysis

The algorithm used by the lifetime analysis pass was proposed by Wimmer and Franz (2010). It exploits certain properties of the SSA representation, which allows the usage of a single reverse iteration over the list of basic blocks, instead of a traditional iterative data-flow analysis. This can be accomplished by using a special block ordering, which has been described above in Section 4.5. When iterated in reverse order, all *usages* of a value are always seen before the definition is reached. Additionally, all blocks belonging to one loop are scheduled contiguously. In contrast to Wimmer and Franz, non-virtual, non-register operands, like fixed registers and stack-slots, are also considered by this implementation. This gives the register allocator a more detailed view of the whole picture. The algorithm computes exact lifetime information where one interval can consist of more than one continuous range with lifetime holes in between. This is especially important for handling fixed registers.

4.8.1 Algorithm

The algorithm computes a lifetime interval for each machine operand. Therefore, a *live set*, containing all operands that are live at the beginning of a basic block, is maintained. The initial content of this set for the current block b is the union of the *liveIn* sets of all successors. Additionally, for each φ-function of a successor, the operand corresponding to b is added. For all operands in this initial live set a live-range, spanning from the start to the end of the block, is created. These ranges are shortened later if needed.

After this initial setup the instructions of b are processed in reverse order. An output operand op is removed from the live set and the start of the range of op is set to the position of the current instruction. Note that the low-level IR is not restricted to single assignments, so this can happen more than once for a given operand op. Analogously an input operand op is added to b's live set and a new range is added from the beginning of b to the current position.

As already mentioned, φ-nodes are not part of the normal instructions and are handled differently. After all instructions of a block b have been processed, the output operand of each φ-function is removed from the live set. The live-ranges are already set to the start of b, so no further adjustments are required. The input operands are handled when their corresponding predecessor is processed as described above.

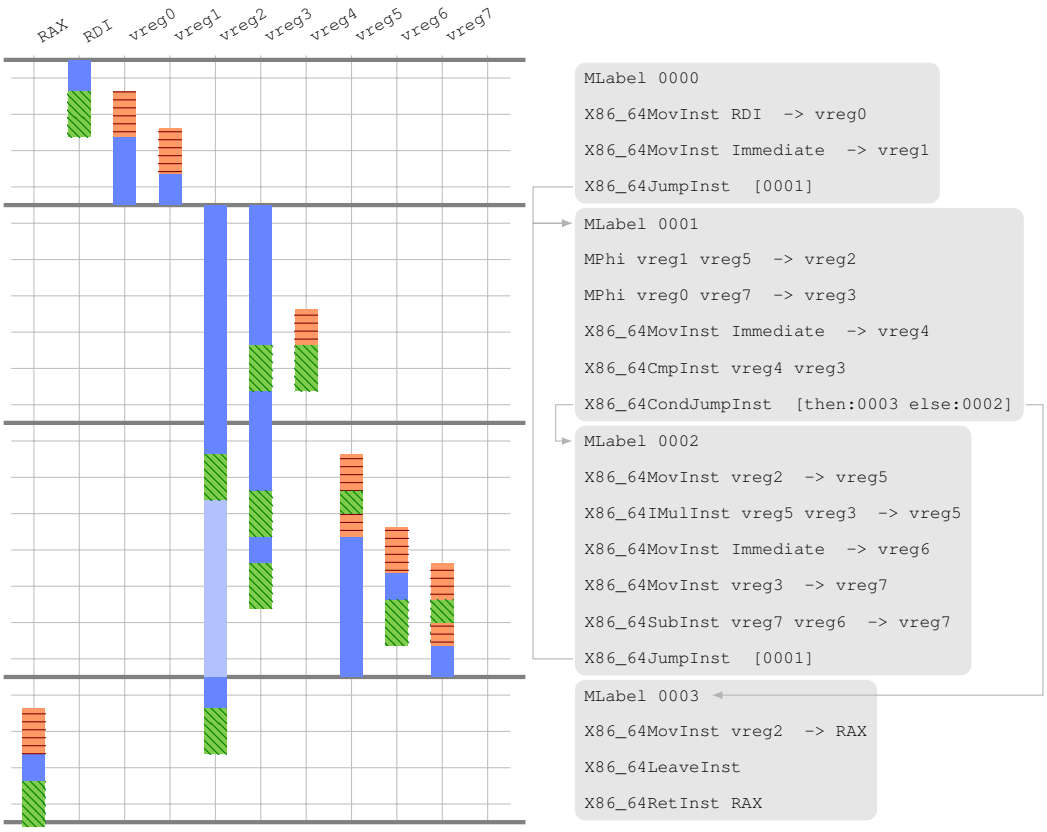


Figure 4.1: Lifetime Interval for Listing 3.1 — Active live-ranges are depicted in darker blue shade. Lifetime holes are colored in light blue. *Def*-positions are orange with horizontal lines, *Use*-positions are green with a diagonal pattern.

For blocks which are the source of a loop back-edge, the live set is not complete at the time of processing. The required information is not available until the live set of the corresponding loop-header block is known. By using the property that all loop blocks are scheduled in one contiguous region, this can be fixed by inserting a live-range from the target of a back-edge to its source. Note that the live sets for loop blocks are not updated and remain incomplete.

Ranges are merged if possible to create continuous intervals. Additionally, the lifetime interval data structure keeps track of use and definition positions. To handle live-ranges not asserted by real usages or definitions, *pseudo use/defs* (Wimmer and Mössenböck, 2005) are created. These are needed for loop back-edges, for lifetime holes and for operands of φ-instructions, which are not considered as real use/def-positions. The result of the lifetime analysis of the program in Listing 3.1 is shown in Figure 4.1. The pseudo code is depicted in Algorithm 4.7.

4.9 Register Allocation

Register allocation is a highly profitable optimization (Poletto and Sarkar, 1999; Traub et al., 1998; Wimmer and Mössenböck, 2005). In the compiler framework it is not only responsible for finding a register assignment for virtual registers, but also performs φ-node resolution and ensures that architecture constraints, such as fixed registers and calling conventions, are adhered.

Classically, register allocation was tackled by calculating the interference graph and solving the *Graph k-coloring Problem* instance, where k is equal to the number of available physical registers. An interference graph is a graph where the nodes represent live-ranges and edges symbolize that two ranges are live at the same time. The k-coloring, for $k \geq 3$, is known to be NP-complete (Karp, 1972), so finding an optimal solution is not tractable in practice. Heuristics have to be used to find a *good enough* solution in *good enough* time. This approach was pioneered by Chaitin et al. (1981); Chaitin (1982). One issue with that method is that the core problem, is not aware of spilling live-ranges. If no solution is found, one or more registers are spilled, that means evacuated on the stack, and the algorithm is restarted. That includes the recreation of the interference graph. Additionally, selecting a node for spilling is not trivial. While improvements to the original approach where proposed, for instance by Briggs et al. (1989) or Choi et al. (1999), the root of the problem, the iterated interference graph construction and solving the k-coloring problem multiple times, remains.

With these deficiencies the graph coloring approach is an unsatisfying candidate for time constraint systems. To overcome these issues Poletto et al. proposed the linear scan register allocator (Poletto et al., 1997; Poletto and Sarkar, 1999). It is fundamentally different to the coloring method. It does not require an interference graph. Instead, registers are allocated in a greedy fashion. The original version of the algorithm does not make use of lifetime holes. Each lifetime interval consists of only one range. All lifetime intervals are traversed in a single linear pass, from the interval with the earliest start position to the interval with the latest. In every iteration a register is selected for the current interval and the interval is added to the *active* set. This set keeps track of all registers currently occupied. At the beginning of each iteration all intervals, which have ended, already are removed from the active set. If there are no registers left, an interval gets spilled. Poletto et al. heuristically chose the interval with the latest end point. Because it is not known at the beginning, if an interval is on the stack or in a register, the instruction operands can not be written in the first pass. Therefore another rewrite phase is needed after the allocation algorithm is done.

Traub et al. (1998) extended the original approach to improve the quality of the generated code. Their algorithm tries to fit small intervals into lifetime holes of longer ones. If there is no lifetime hole which is long enough, they split an existing interval and insert a move to the stack. If at any iteration the current instruction uses a variable, which is currently on the stack, the algorithm searches for a free register, possibly spilling the interval occupying this register, and loads the value from the stack. Afterwards the value is not immediately moved back to the stack but stays in the new registers as long as no other interval requires it. Because a lifetime interval can switch its position from register and stack back and forth, this algorithm is also called *second-chance binpacking*. The ability to split and spill intervals is also used to handle usage conventions for example caller saved registers. Function calls impose a barrier for lifetime

intervals and force the algorithm to split and evict the value onto the stack. In other systems this is done by a separate pass but with this algorithm the functionality is already included.

In contrast to the approach by Poletto and Sarkar, where the algorithm iterates over the lifetime intervals, Traub et al. need to consider every use/def position, which basically means to iterate over all instructions. Another difference is that the operands of an instruction will never change once the instruction has been processed, therefore the rewriting can be performed in the main pass of the algorithm.

The fact, that a single interval may be stored in different places, can introduce inconsistencies on basic block boundaries. A *resolution* pass is needed to reestablish the semantic correctness. For each control-flow edge it checks the location of an interval at the end of the predecessor and at the beginning of the successor. If both locations differ, a *move* instruction is inserted. Traub et al. (1998) already mentioned that this resolution shares properties of SSA deconstruction algorithms. A property that is exploited by the approach used in this work.

Wimmer and Mössenböck (2005) described the register allocator developed for the HotSpot Client Compiler. They follow the direction of Traub et al. but propose extensions to improve the result for their target language and architecture. In contrast to Traub et al. their approach iterates over lifetime intervals, which are equipped with exact use/def markers. This way the algorithm is independent of the intermediate representation. Additionally, these use/def positions are marked whether they *must* or *should* be in a register. The idea is to make use of addressing capabilities found in CISC architectures like x86_64. If an instruction at an use/def position can address an operand from memory, it uses the stack-slot of the spilled interval directly without moving it into a register. Wimmer and Mössenböck continue the idea of handling system constraints using the register allocator. They create *fixed intervals* to model restrictions like calling conventions. A fixed interval must not be touched by the register allocator but used to determine the set of available registers. *Register hints* are introduced to eliminate redundant moves. If two intervals are connected only by a *move*, the allocator tries to assign both intervals to the same register, even if the heuristic would suggest a different solution. These hints do not require any additional analysis and can be calculated during liveness analysis. Empirical evaluation suggests that this simple approach can reduce the number of register to register moves by 25% average. In contrast to Traub et al. instruction rewriting is performed in a separate pass. This allows flexible placement of spill instructions and is used for optimizations. For example moves are placed outside of loops or at basic block boundaries. The former reduces the number of *executed* instructions, whereas the later can *prevent the creation of moves* during the resolution phase.

Later Wimmer and Franz (2010) proposed, that register allocation should be performed directly on SSA Form. This extension to the original algorithm (Wimmer and Mössenböck, 2005) allows the use of the simple, still exact lifetime analysis pass described in Section 4.8. This makes the implementation of a separate SSA deconstruction unnecessary. As Traub et al. already remarked, φ-node deletion and the resolution needed due to CFG corruption are similar operations and can be performed simultaneously.

4.9.1 Algorithm

The register allocator implemented in course of this work is based on the approach by Wimmer and Mössenböck (2005) with the extensions proposed in Wimmer and Franz (2010). The pseudo

code is shown in Algorithm 4.9. It maintains four sets that represent the current state of the computation. The unhandled set contains all unassigned intervals sorted by increasing starting position. If an interval is currently live, it is stored in the active set. Intervals that have a lifetime hole at the current position are stored in the inactive set. If an interval has already ended, it is stored in handled. The unhandled set is initialized to the union of all lifetime intervals, all other sets are empty at the beginning. At each iteration of the main loop one element of unhandled is popped until the set is empty. First, all intervals in active and inactive are checked if their state changes. Once these sets are up to date, the algorithm tries to find a free register using the method try_allocate_free(), which is depicted in Algorithm 4.10.

For each physical register, free_until_pos is calculated. This is the position when a register is used for the next time after the current position. The register with the highest free_until_pos is selected as allocation candidate. If the current interval ends before the allocation candidate is needed again the allocation succeeded and the candidate is assigned to the current interval. If the current interval is required longer than the candidate is available, current is split right before the free_until_pos of the candidate. The stripped interval of current is assigned to the selected register and the allocation succeeds. The remaining part of the interval is inserted into unhandled and will be processed later.

In case no register is available, the method fails and passes control to allocate_blocked(). This method, which is shown in Algorithm 4.11, has the duty to select an interval for spilling, therefore the next_use_pos is calculated. The interval with the highest next_use_pos is selected for spilling. This even can be the current interval. In this case current is spilled to the stack and the interval is split before the first use position. If a different interval has been selected this interval is spilled at the current position and current is assigned to the register candidate. Additionally, all inactive intervals, that use the same register, are spilled after the end of their lifetime hole.

Fixed intervals are handled differently than proposed in the original paper. Wimmer and Mössenböck check if the current interval intersects with a fixed interval at the end of allocate_blocked(). In this implementation fixed intervals are included in the state sets. Therefore such an intersection can not happen. Also note that fixed intervals are never selected for spilling.

After all intervals are allocated, resolve() takes care of CFG inconsistencies and φ-functions. The pseudo code is depicted in Algorithm 4.8. To do so, all CFG edges are traversed. If the store for a lifetime interval at the end of the predecessor is different to the store at the beginning of the successor, a *move* is created. Unfortunately these moves have interdependencies that need to be taken care of. The dependency graph can contain cycles, which have to be broken by introducing a new register. The same is needed for stack-to-stack moves, which are not supported directly by most architectures.

The pseudo codes for the Algorithms 4.9-4.11 are taken from Wimmer and Mössenböck (2005), Algorithms 4.7-4.8 from Wimmer and Franz (2010).

Some of the differences are described in the following. In contrast to Wimmer and Mössenböck spill code is inserted during the linear scan pass, as it is done in the implementation by Traub et al.. This has the advantage that the exact position is known, and a new interval for a *virtual stack-slot* can be created. Virtual stack-slots are analog to virtual registers and must be assigned to a physical stack offset. Unused stack-slots are reused, which shrinks the size of the

```
1  Procedure build_intervals()
2      foreach block b in reverse order do
3          foreach successor of b do live ← union of successor.liveIn
4          foreach phi function phi of successors of b do
5              live.add(phi.inputOf(b))
6          end
7          foreach opd in live do
8              intervals[opd].addRange(b.from, b.to)
9          end
10         foreach operation op of b in reverse order do
11             foreach output operand opd of op do
12                 intervals[opd].setFrom(op.id)
13                 live.remove(opd)
14             end
15             foreach input operand opd of op do
16                 intervals[opd].addRange(b.from, op.id)
17                 live.add(opd)
18             end
19         end
20         foreach phi function phi of b do
21             live.remove(phi.output)
22         end
23         if b is loop header then
24             loopEnd ← last block of the loop starting at b
25             foreach opd in live do
26                 intervals[opd].addRange(b.from, loopEnd.to)
27             end
28         end
29         b.liveIn ← live
30     end
```

Algorithm 4.7: Build Lifetime Intervals

activation record. Because stack-slots are modeled similarly to registers, the register allocation algorithm can be used. This comes to the expense of managing larger state sets during allocation. Paleczny et al. (2001) mentioned that a similar approach is used in the HotSpot server compiler.

4.10 Code Emission

The code emission pass consists of 3 phases. Instruction *emission*, instruction *linking* and code *installation*.

Instruction emission is done using a single reverse traversal over the MachineInstructions of a method. The CodeMemory is a temporary store for the machine code. It consists of two segments, the CodeSegment and the DataSegment. The code segment contains the machine instructions in binary form. The data segment is used for constant values but also for other data that is subject to patching, like the address of a call target. These addresses can change after the compilation because either the method is not yet known so not call target exists, or the method is recompiled and the old code is no longer valid. Instead of hardcoding the method address into

```
1  Procedure resolve()
2      foreach control flow edge from predecessor to successor do
3          foreach interval it live at begin of successor do
4              if it starts at begin of successor then
5                  phi ← phi function defining it
6                  opd ← phi.inputOf(predecessor)
7                  if opd is a constant then
8                      moveFrom ← opd
9                  else
10                     moveFrom ← location of intervals[opd] at end of predecessor
11                 end
12             else
13                 moveFrom ← location of it at end of predecessor
14             end
15             moveTo ← location of it at begin of successor
16             if moveFrom ≠ moveTo then
17                 mapping.add(moveFrom, moveTo)
18             end
19         end
20         mapping.orderAndInsertMoves()
21     end
```

Algorithm 4.8: Resolution

the machine code, the value is read from the data segment. This approach releases the patcher from the burden of modifying binary instruction codes, which can cause problems regarding multi-threading and instruction caches. In contrast, data segment modifications are atomic.[2] For writing the machine code each MachineInstruction contains a member function emit, which is responsible for emitting the binary representation of the instruction. Due to the reverse order, the targets of forward jumps are known and are already inserted. Especially fall-through jumps, that are jumps to the next instruction, can be eliminated easily.

Backward jumps and references to the data memory need a second pass to determine their final position. This is done in the instruction linking phase. To inform the code emission pass that a MachineInstructions requires linking, it inserts itself into a link-later list in the emit method. To define the linking action a MachineInstruction overwrites the link member. The CodeMemory supports the linking process by providing symbolic labels to both, code segment and data segment entries.

In the last phase of the pass, the temporary CodeMemory is moved to the final position. Therefore the addresses used to target entries inside a method are relative. In other words, a method is position independent code (PIC). In addition to the copying, the last phase marks the new code block executable and informs the virtual machine run-time about the new method position.

[2]The way to achieve this is target dependent. In the context of the x86_64 architecture, memory stores to quadword-aligned addresses are atomic (Int, 2013, Section 8.1.1).

```
1  Procedure linear_scan()
2      while unhandled ≠ ∅ do
3          current ← pick and remove first interval from unhandled
4          position ← start position of current
           // check for intervals in active that are handled or inactive
5          foreach interval it in active do
6              if it ends before position then
7                  move it from active to handled
8              else if it does not cover position then
9                  move it from active to inactive
10             end
11         end
           // check for intervals in inactive that are handled or active
12         foreach interval it in inactive do
13             if it ends before position then
14                 move it from inactive to handled
15             else if it covers position then
16                 move it from inactive to active
17             end
18         end
           // find a register for current
19         try_allocate_free(current)
20         if allocation failed then allocate_blocked(current)
21         if current has a register assigned then add current to active
22     end
```

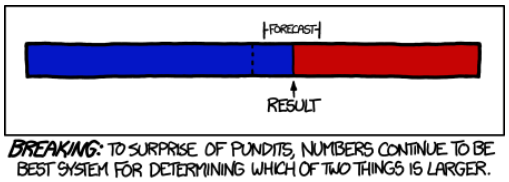
Algorithm 4.9: Linear Scan Algorithm

```
1  Procedure try_allocate_free(current)
2      set freeUntilPos of all physical registers to maxInt
3      foreach interval it in active do
4          freeUntilPos[it.reg] ← 0
5      end
6      foreach interval it in inactive intersecting with current do
7          freeUntilPos[it.reg] ← next intersection of it with current
8      end
9      reg ← register with highest freeUntilPos
10     if freeUntilPos[reg] = 0 then
           // no register available without spilling
11         allocation failed
12     else if current ends before freeUntilPos[reg] then
           // register available for the whole interval
13         current.reg ← reg
14     else
           // register available for the first part of the interval
15         current.reg ← reg
16         split current before freeUntilPos[reg]
17     end
```

Algorithm 4.10: Try Allocate Free

```
1  Procedure allocate_blocked(current)
2      set nextUsePos of all physical registers to maxInt
3      foreach interval it in active do
4          nextUsePos[it.reg] ← next use of it after start of current
5      end
6      foreach interval it in inactive intersecting with current do
7          nextUsePos[it.reg] ← next use of it after start of current
8      end
9      reg ← register with highest nextUsePos
10     if first usage of current is after nextUsePos[reg] then
           // all other intervals are used before current,
           // so it is best to spill current itself
11         assign spill slot to current
12         split current before its first use position that requires a register
13     else
           // spill intervals that currently block reg
14         current.reg ← reg
15         split active interval for reg at position
16         split any inactive interval for reg at the end of its lifetime hole
17     end
       // make sure that current does not intersect with
       // the fixed interval for reg
18     if current intersects with the fixed interval for reg then
19         split current before this intersection
20     end
```

Algorithm 4.11: Allocate Blocked



As of this writing, the only thing that's 'razor-thin' or 'too close to call' is the gap between the consensus poll forecast and the result.

CHAPTER 5

Evaluation

To get an indication for the practical performance of the implementation, the second-stage compiler was compared with the baseline compiler.

5.1 Methodology

5.1.1 Test System

All tests where performed on an *Intel® Core™ i7 CPU M 620* processor at 2.67GHz with 2 cores and 4 threads. The system comprises 8GiB main memory running on a GNU Linux operating system in 64bit mode with a *3.5.0-42* kernel.

5.1.2 Configuration

CACAO was configured for a release build[1] and to use GNU Classpath. It was compiled using a recent LLVM Clang compiler (pre-3.4) with optimizations enabled[2]. Besides the second-stage compiler[3], the statistics[4], timing[5] and logging[6] modules were enabled.

5.1.3 Benchmarks

Popular benchmark suits, such as SPECjvm (Shiv et al., 2009) or DaCapo (Blackburn et al., 2006), where not applicable because 1.) at the time of writing the implementation was not advanced enough to run these suits, and 2.) the interference of other virtual machine components

[1] `--disable-debug`
[2] `--enable-optimizations`
[3] `--enable-compiler2`
[4] `--enable-statistics`
[5] `--enable-rt-timing`
[6] `--enable-logging`

make it difficult to identify the direct influence of the compiler. Instead, *micro-benchmarks* where used to compare both implementations. The presented values are an arithmetic mean of 30 runs.

5.2 Results

5.2.1 Comparison to the Baseline Compiler

Figure 5.1 illustrates the compilation time differences between the baseline and the second-stage compiler. For given tests, the new implementation is about 30–50 times slower than the existing compiler. Although the new compiler was not designed to be as fast as the baseline compiler, the gap is higher than expected. Presumably, the reason lays in the fact, that the second-stage compiler has not yet been optimized towards compilation speed and still contains test and debug code. Nevertheless, the exact reasons for the moderate performance have to be investigated. Figure 5.4 depicts the breakdown of the compilation time of the second-stage compiler to the individual passes. More than 50% of the time is spent in the lifetime analysis and the register allocator. These are the most complex passes in the pipeline and should receive the most optimization efforts in the near future. The numbers for the compilation time diagram are listed in Table 5.1 and Table 5.2.

Figure 5.2 depicts the resulting code sizes for both compilers. Although the second-stage compiler performs no optimizations beside instruction scheduling and linear-scan register allocation, it can compete with the code size generated by the baseline compiler. Note that the new implementation currently does not even support code generation optimization, like coding immediate values directly into the instruction opcodes, nor are the operands loaded directly from the stack. All values are loaded into registers similar to RISC-like load-store architectures. This suggests that there is high optimization potential.

In Figure 5.3 the run time for each benchmark is illustrated. It can be seen that the code generated by the new compiler is almost as fast as the baseline compilers code. The limitations regarding code generation described above also apply to the run time behavior.

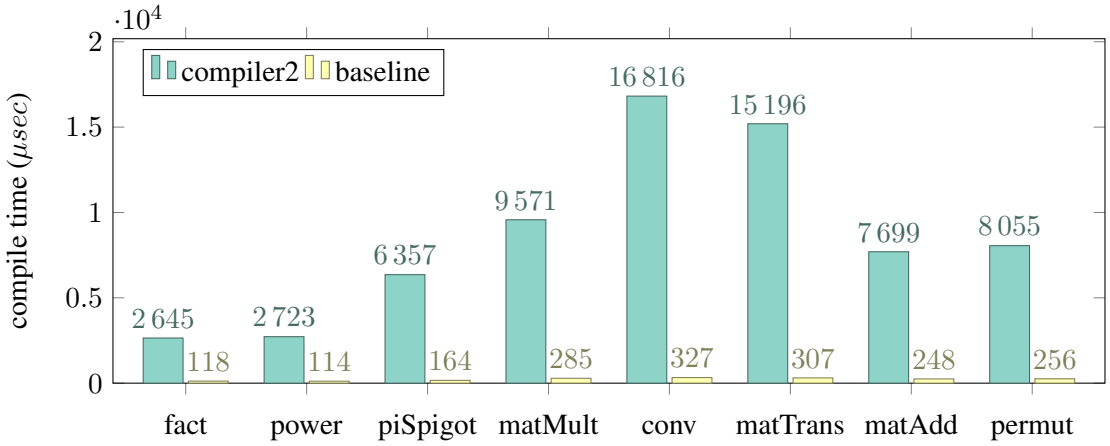


Figure 5.1: Compilation Time

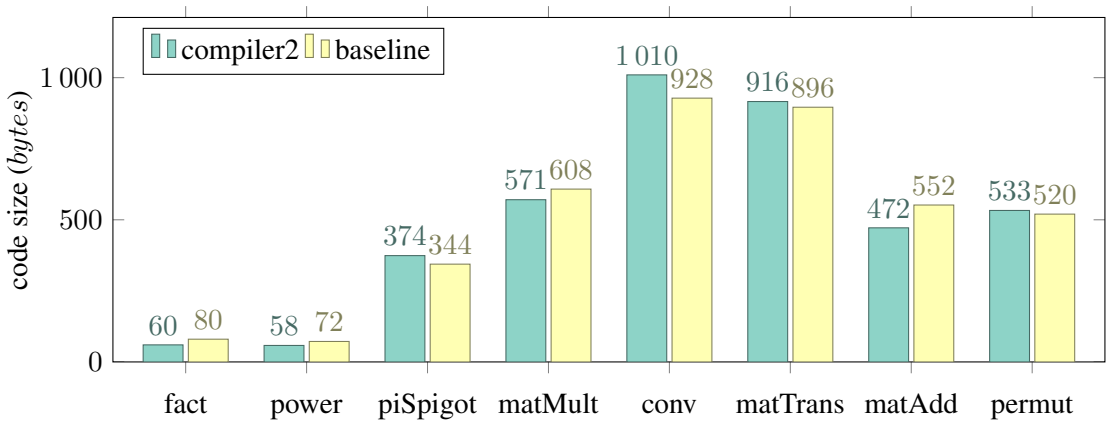


Figure 5.2: Code Size

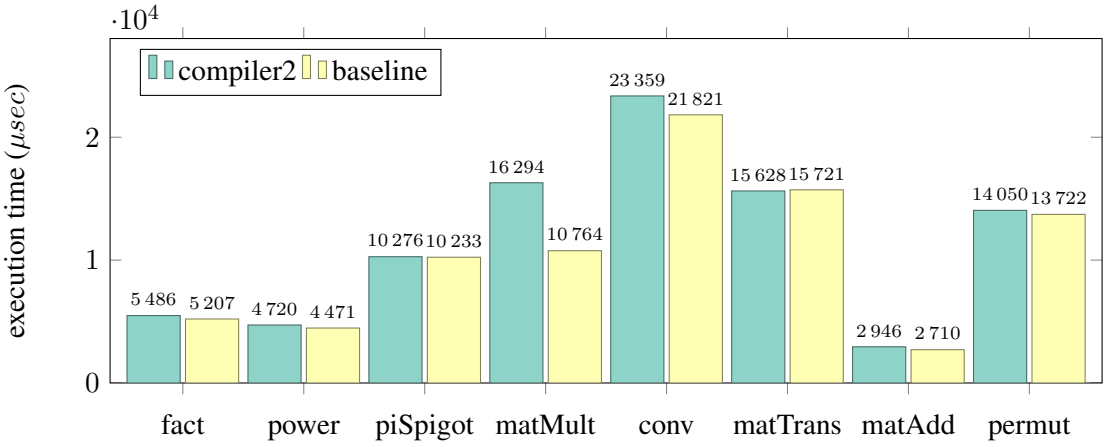


Figure 5.3: Execution Time

	Compile time (μsec)			Code size ($bytes$)		%
Benchmark	Compile2	Baseline	$c2/bl$	Compiler2	Baseline	$c2/bl$
matAdd	7 699	248	31.1	472	552	85
fact	2 645	118	22.4	60	80	75
conv	16 816	327	51.4	1 010	928	109
piSpigot	6 357	164	38.7	374	344	109
matTrans	15 196	307	49.4	916	896	102
permut	8 055	256	31.4	533	520	102
power	2 723	114	23.8	58	72	81
matMult	9 571	285	33.6	571	608	94

	Execution time (μsec)		%
Benchmark	Compile2	Baseline	$c2/bl$
matAdd	2 946	2 710	109
fact	5 486	5 207	105
conv	23 359	21 822	107
piSpigot	10 276	10 233	100
matTrans	15 628	15 721	99
permut	14 050	13 722	102
power	4 720	4 471	106
matMult	16 294	10 764	151

Table 5.1: Comparison Baseline and Compiler2

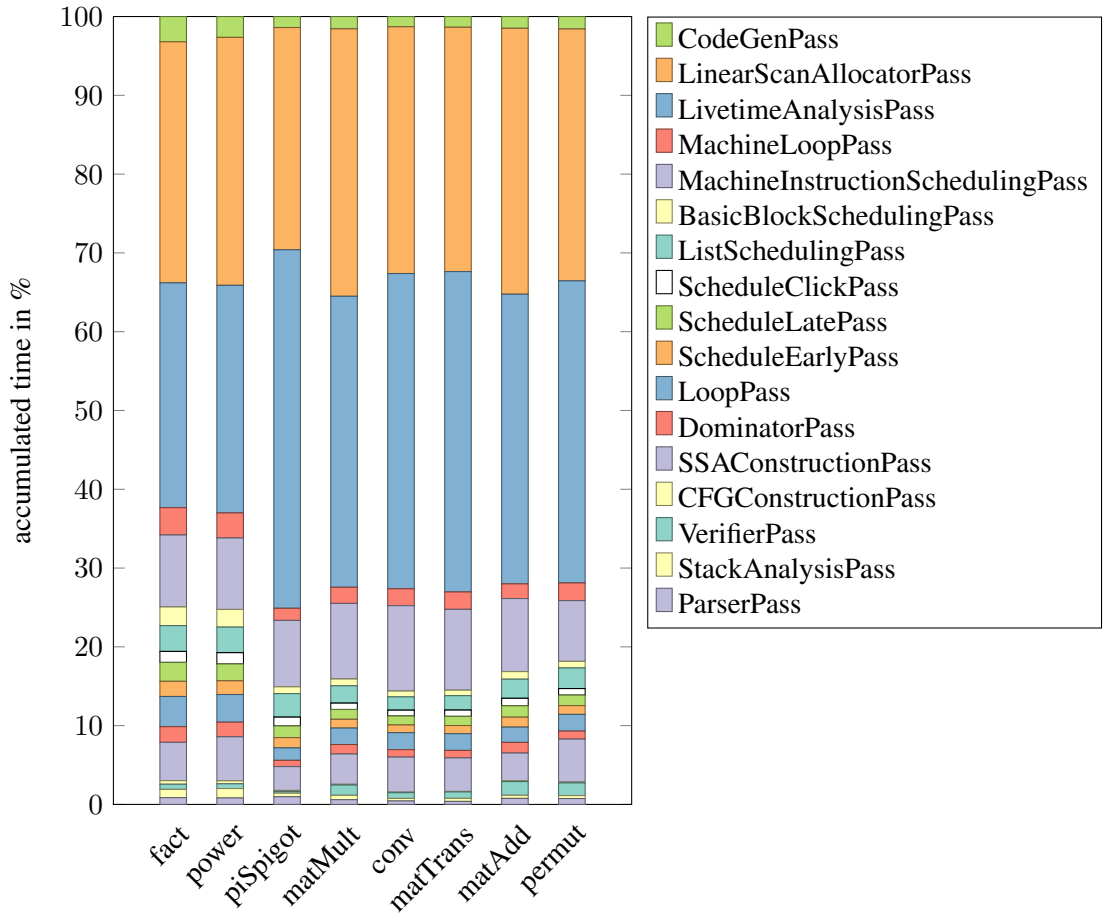


Figure 5.4: Compiler2 Compilation Time per Pass

Pass	matAdd	fact	conv	piSpigot	matTrans	permut	power	matMult
ParserPass	0.8	0.9	0.4	1.0	0.4	0.7	0.8	0.6
StackAnalysisPass	0.4	1.0	0.3	0.5	0.4	0.4	1.2	0.6
VerifierPass	1.7	0.7	0.7	0.2	0.8	1.6	0.6	1.3
CFGConstructionPass	0.1	0.4	0.1	0.1	0.1	0.1	0.4	0.1
SSAConstructionPass	3.5	4.9	4.4	3.0	4.3	5.5	5.6	3.8
DominatorPass	1.4	2.0	0.9	0.8	1.0	1.0	1.9	1.2
LoopPass	1.9	3.8	2.2	1.6	2.1	2.1	3.5	2.1
ScheduleEarlyPass	1.3	1.9	1.0	1.3	1.0	1.1	1.7	1.1
ScheduleLatePass	1.4	2.4	1.1	1.5	1.2	1.3	2.1	1.2
ScheduleClickPass	1.0	1.4	0.8	1.1	0.8	0.8	1.4	0.8
ListSchedulingPass	2.4	3.3	1.7	3.0	1.8	2.6	3.2	2.2
BasicBlockSchedulingPass	1.0	2.4	0.7	0.9	0.7	0.8	2.2	0.9
MachineInstructionSchedulingPass	9.3	9.1	10.8	8.4	10.3	7.7	9.1	9.6
MachineLoopPass	1.9	3.5	2.2	1.5	2.2	2.2	3.2	2.1
LivetimeAnalysisPass	36.8	28.5	40.0	45.5	40.7	38.3	28.9	36.9
LinearScanAllocatorPass	33.8	30.6	31.3	28.2	31.0	32.0	31.5	33.9
CodeGenPass	1.5	3.2	1.3	1.4	1.4	1.6	2.6	1.6

Table 5.2: Compiler2 Compilation Time per Pass (in %)

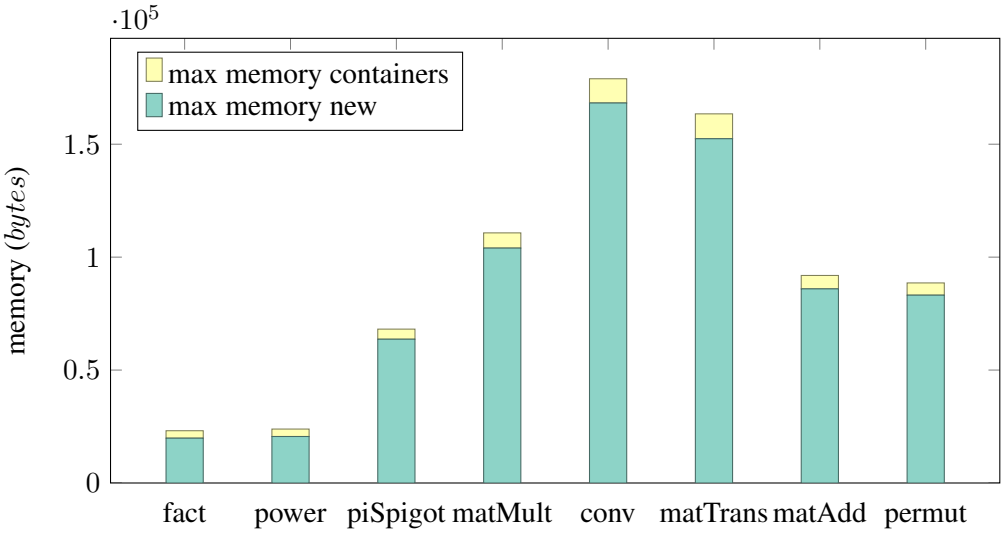


Figure 5.5: Compiler2 Memory

5.2.2 Dynamic Memory Consumption

The dynamic heap memory allocated by the compiler can be divided into two categories. First, the memory explicitly allocated using the new operator. Second, the data stored in standard library containers. Figure 5.5(a) shows that the first category is responsible for the major part of the memory usage. This is reasonable, because most containers only store pointers.

The relation between the benchmarks is similar to the relation of the according code sizes, what suggests that there is no exponential explosion in term of heap memory consumption.

For deeper insight, the behavior of each class has been analyzed. Figure 5.6 lists the top five consumers. Most memory is used to store MachineOperands, like (virtual) registers and stack slots. The operands introduced to handle two-address instructions and fixed register ranges are mainly responsible for this fact. The LIR instructions also demand a great part of the heap memory. The x86_86 instruction set and its representation in the target implementation are to blame for this. A single native instruction allows many different addressing and operand modes. This makes MachineInstructions complicated and heavy-weighted regarding memory requirements. Creating different specialized versions for each instruction can reduce the impact of this issue. The other three large scale consumers are HIR instructions, lifetime intervals and machine basic blocks. Currently, most memory is used for low-level operations. This might change as soon as HIR optimizations are added to the compiler.

5.2.3 Linear Scan Allocator

Some characteristic numbers of the register allocator and resolution pass are depicted in Figures 5.7-5.10. *Spill stores* are the number of values evicted on the stack, whereas *spill loads* denote stack loads. Note that the allocator outsources many operations related to lifetime split-

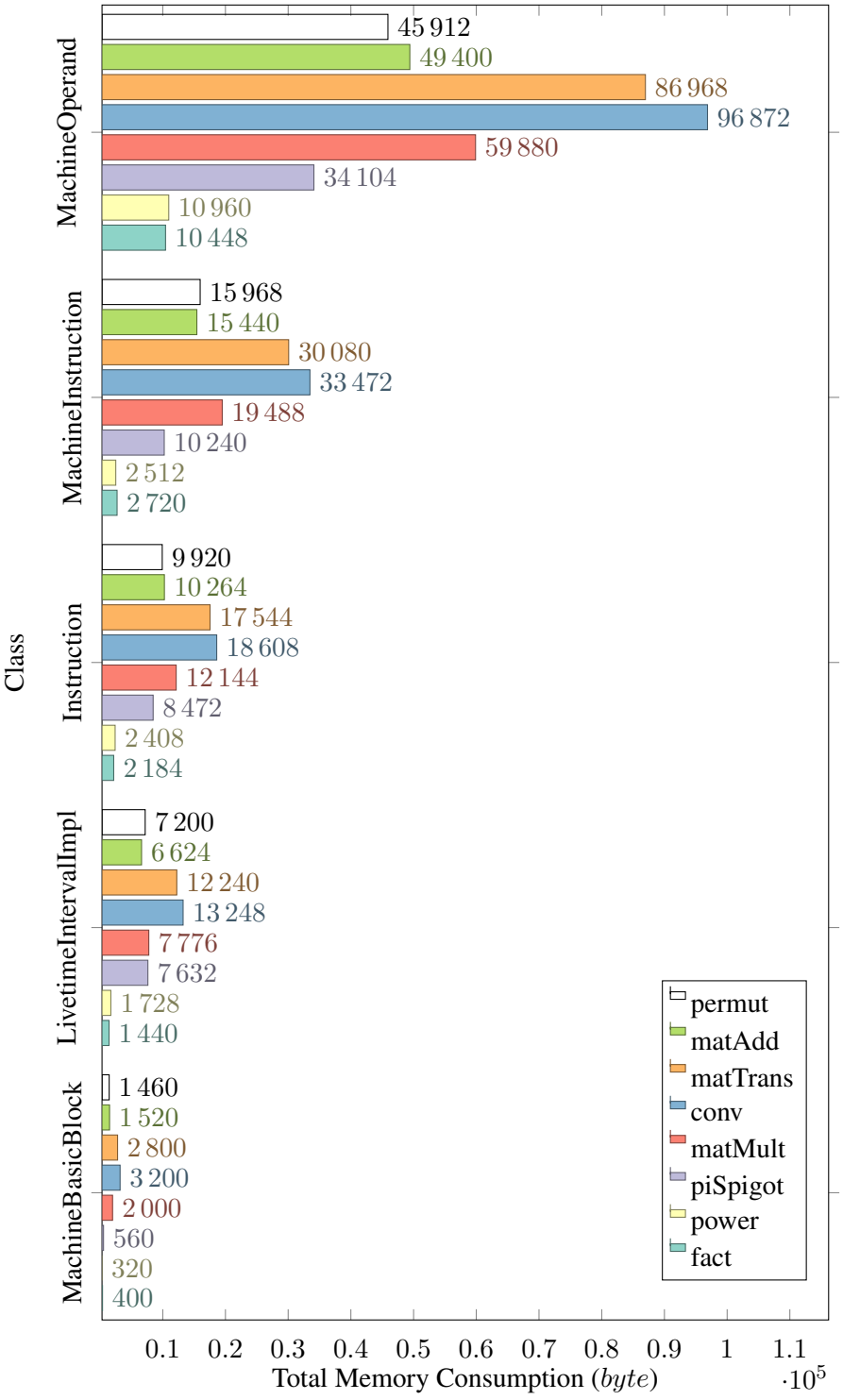


Figure 5.6: Memory Consumption per Class

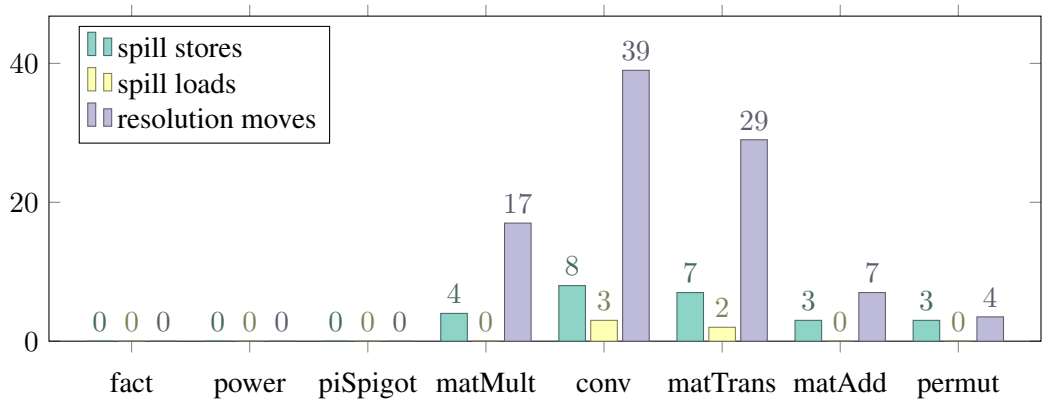


Figure 5.7: Linear Scan Statistics (spills)

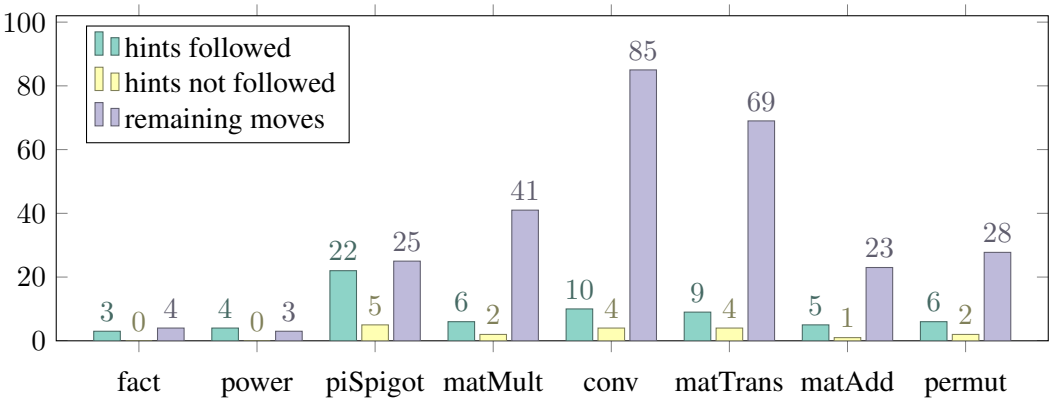


Figure 5.8: Linear Scan Statistics (hints)

ting to the resolution algorithm. These numbers are included in *resolution moves* together with the moves required for φ-node deconstruction. Figure 5.8 shows that most register hints are taken into account. All emitted move instructions are collected in the *remaining moves* entry. This number includes constant loads and stack moves. The high number registers available for register allocation and utilization of lifetime holes, render interval splitting a rare event, which is illustrated by the difference between *allocate free* and *allocate blocked* in Figure 5.9. *Resolution reg* and *resolution stack* denotes the operands required to resolve circular dependencies and stack-to-stack moves in the resolution phase. All numbers are summarized in Table 5.3-5.6.

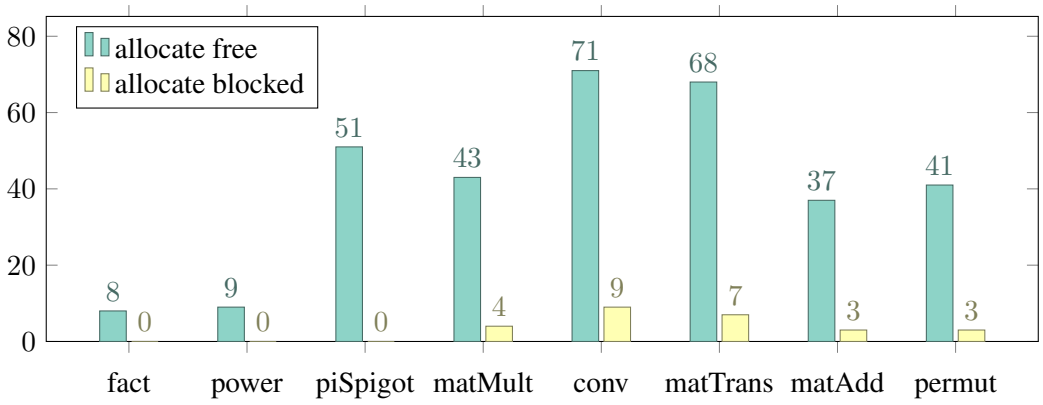


Figure 5.9: Linear Scan Statistics (free/blocked)

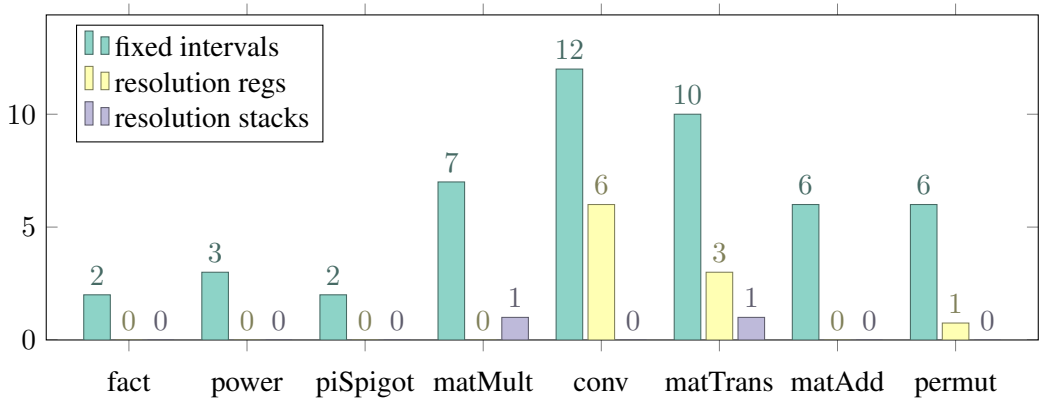


Figure 5.10: Linear Scan Statistics (resolution)

Benchmark	spill stores	spill loads	resolution moves
matAdd	3	0	7
fact	0	0	0
conv	8	3	39
piSpigot	0	0	0
matTrans	7	2	29
permut	3	0	4
power	0	0	0
matMult	4	0	17

Table 5.3: Linear Scan Statistics (spills)

Benchmark	hints followed	hints not followed	remaining moves
matAdd	5	1	23
fact	3	0	4
conv	10	4	85
piSpigot	22	5	25
matTrans	9	4	69
permut	6	2	28
power	4	0	3
matMult	6	2	41

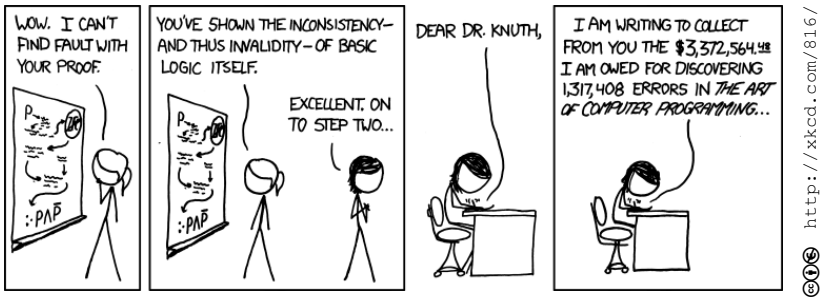
Table 5.4: Linear Scan Statistics (hints)

Benchmark	allocate free	allocate blocked
matAdd	37	3
fact	8	0
conv	71	9
piSpigot	51	0
matTrans	68	7
permut	41	3
power	9	0
matMult	43	4

Table 5.5: Linear Scan Statistics (free/blocked)

Benchmark	fixed intervals	resolution regs	resolution stacks
matAdd	6	0	0
fact	2	0	0
conv	12	6	0
piSpigot	2	0	0
matTrans	10	3	1
permut	6	1	0
power	3	0	0
matMult	7	0	1

Table 5.6: Linear Scan Statistics (resolution)



Dear Reader: Enclosed is a check for ninety-eight cents. Using your work, I have proven that this equals the amount you requested.

CHAPTER 6

Critical Reflection

6.1 Redundancies

The redundancy between the baseline compiler and the new compilation framework is one problem with the current state of affairs.

Intermediate Representation

This work introduced two more intermediate representations. Together with the baseline IR, three different forms of a program exist in the CACAO VM. These representations are rather different from each other, what makes sharing code and knowledge between them more complicated. Furthermore, for deoptimization and adaptive recompilation the virtual machine must translate between these representations.

Target Implementation

Implementing the machine code emitter twice for each architecture is tedious and error prone. During the design process, it was considered to translate the IR back into the baseline IR at some point. This approach was rejected later because 1.) it would be too costly and complex to set up all required data structures with their implicit dependencies, and 2.) the framework would lose much of its potential due to the different abstraction levels of the low-level and the baseline IR.

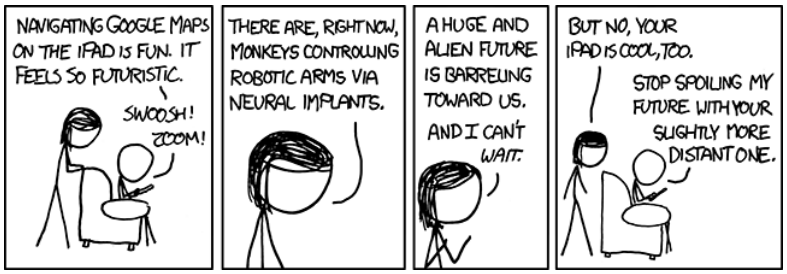
Using only the existing code generator was also not feasible, because it is interwoven with the baseline compiler. Breaking these interdependencies would have meant major refactoring of the existing passes, which was negative assertion in the first place. Section 7.1.1 proposes a possible solution to this problem.

6.2 Third-Party Library Support

As already stated in the introduction, it was an explicit design goal to avoid new third-party libraries. Additionally, new code for the CACAO VM still must conform to the out-dated C++03 standard (ISO, 2003). In retrospect these decisions made the implementation unnecessary complicated and constrained the work. Section 7.1.2 advocates the use of the Boost C++ libraries (Schling, 2011) and the recent C++11 standard ISO (2012b) for future projects.

6.3 Compiler Performance

Chapter 5 showed that the performance leaves much to be desired. While there is optimization potential the question arises as to whether some design decisions are simply not adequate for a just-in-time compiler. Another point that needs consideration is memory consumption. Detailed investigations still needs to be done, but estimations suggest that the peak and dynamic heap memory consumption exceed the baselines requirements. The baseline compiler uses a *dump memory*. All dynamically allocated data is located in one continuous memory block, which is allocated before the compiler executes and is deleted afterwards. Therefore no memory is freed during compilation, so interaction with the memory allocator is rendered unnecessary. For the second-stage compiler this approach is not viable, because it makes heavy use of standard library containers, which in turn often allocate and deallocate memory. This exceeds the dump memory rather quickly.



*Maybe we're all gonna die, but we're gonna die in *really cool ways*.*

CHAPTER 7

Future Work and Summary

This chapter presents possible future work, dicusses alternative approaches and provides a short summary of this paper.

7.1 Future Work

The current implementation can be seen as a proof-of-concept prototype and still lacks many features until it reaches an alpha status.

To date about 60% of the baseline IR instructions are translated into HIR. The most of the remaining instructions can be implemented easily. The only reason why they are not yet handled, is that they do not occur in any of the test cases. The lowering of instructions is more difficult. About half of all implemented HIR instructions (so 30% of the baseline IR) are currently supported by the x86_64 target implementation. The complicated instruction set architecture (ISA) of this processor family poses the main difficulty in this task. Section 7.1.1 proposes an alternative approach to decouple this from the compiler and in general from the CACAO VM.

Besides the missing features of the compiler implementation, the most important future goal is to rework the on-stack replacement mechanism, meaning a generalization of the work of Steiner (2007), to make the implementation compiler independent. Especially replacement points and machine to source state translation are challenging. Also, second-stage compiler guards must be considered. In general the baseline compiler can not provide a replacement point for each position, where a guard might be inserted. Reinvoking the baseline compiler to create the required replacement points *lazily* is one solution but this would lead to many different code versions.

With continuous recompilation, code memory evolution becomes more important. Solutions are needed to handle multiple version of the same method with different assumptions. Additionally, unused code must be freed to keep the code memory in bound.

To guide the compiler with run-time information a profiling framework is needed. The idea is to present this data in an uniform way to the compiler, independent from the profiling approach. These profiling strategies include intrusive instrumentation and stack sampling, but also heuristics, such as the method proposed by Ball and Larus (1996), are possible.

7.1.1 Optimization and Alternative Solutions

SSA Construction Improvements

Directly Create High-level IR

Generating the high-level IR directly from JVM bytecode would avoid the detour of creating the baseline IR. This approach provides major challenges.

- It would require an independent bytecode verifier (Coglio, 2003). This might be circumvented by arguing that the second-stage compiler only compiles bytecode the baseline compiler has already processed and therefore verified already.
- Because the JVM is a stack machine a stack-slot to variable translation is required.
- The JVM bytecode is not fully typed, so a type checker is required.

In the end this means a lot of duplicated functionalities and the efforts needed for implementation and maintenance seem not to be worth it. Additionally, the numbers in Table 5.2 suggest that the baseline IR creation is not the bottleneck of the pipeline.

Parse-time Optimizations

Braun et al. (2013) suggest additional parse-time optimizations, which involve zero or only minor overhead. They perform *arithmetic simplification* as a peephole optimization to decrease the number of created IR instructions. A related operation is constant folding, where constant expressions are evaluated at compile time and replaced with the constant result. *Common subexpression elimination* reuses the value numbering already needed for the SSA construction. The current implementation already "performs" *copy propagation*. The absence of *copy* or *move* constructs in the IR representation implies this.

Although all these optimizations can be easily done during the SSA construction, they are still useful as ordinary compiler passes, because program transformations may introduce new improvement possibilities.

As already mentioned above, the current implementation does not yet create minimal SSA form for irreducible control-flow graphs. That means that there are superfluous φ-nodes in the resulting HIR. Braun et al. proposed a solution to this issue.

Assumption in the HIR

Assumptions are an important tool to extent the scope on which optimizations can be applied. Paleczny et al. (2001) proposed that this information should be a central part of the intermediate representation. For example, if an object is created via its constructor, the static type is known

without class hierarchy analysis. Similarly, guard instructions could be inserted to establish certain constraints. It needs further investigation to find a compact representation of assumptions, so that optimization passes can access this information easily.

Register Allocator

Linear Scan Improvements

As mentioned by Wimmer and Franz (2010), moving out of SSA during the register allocation is not essential. Alternatively φ-functions could be inserted for spilled intervals. Consequently resolution must not be performed in course of register allocation. The φ-functions would take care of CFG inconsistencies. Additionally, the nonlinear part in the algorithm by Wimmer and Mössenböck (2005) can be completely omitted. It seems that this could decrease the complexity of the current approach. It still has to be evaluated if the construction of additional φ-nodes does not cancel these benefits.

Different Register Allocation Approaches

Olesen (2011) highlighted some deficiencies with the linear-scan algorithm in practical implementations and introduced a new greedy register allocator for LLVM.

Lifetime Analysis

The lifetime analysis used in this implementation calculates exact liveness information. Alternative methods, for instance as proposed by Probst et al. (2002), approximate this analysis by asserting safe assumptions. The approach is especially interesting as it can deal with incomplete information, a feature that may gain importance when deoptimization, inlining and assumptions become available.

Tree Pattern Matching Instruction Selection

Currently, lowering is done for each HIR instruction independently. A tree pattern matching algorithm, for example the approach described by Fraser et al. (1992b,a), would certainly achieve better results. There are two challenges with this approach. First, machine state to source state translation gets more difficult, because every value in the HIR graph can no longer be directly matched to the result of one instruction in the LIR. Second, the pattern-matcher is either completely target dependent or a target independent implementation requires detailed information from the architectural part. A semi-automatic target generation approach, such as suggested in Section 7.1.1, could make this feasible.

Even more powerful approaches have been proposed, which directly work on the SSA graph. The method by Ebner et al. (2008) can match the complex DAG[1] patterns, like SIMD[2] instructions. Unfortunately, the matching problem becomes NP complete when extended to such patterns. To overcome this issue, Ebner et al. used a PBQP solver to compute the result. While the

[1] Directed Acyclic Graph
[2] Singe Instruction Multiple Data

compile time overhead is acceptable for ahead-of-time compilation, this solution is not tractable for a JIT system.

Instruction Scheduling on the Low-level IR

Instruction scheduling is currently performed on the HIR. LIR scheduling could profit form the information provided by the target implementation such as delay slots. Because of the structure of the LIR, this requires a new data-flow analysis to build the data-dependence graph for machine operands. Additionally, scheduling can not completely be postponed to the low-level representation, because some HIR optimizations also require a valid schedule.

Shared Graph Template Framework

Many graph algorithms occur multiple times throughout the implementation. For example list scheduling is performed by the instruction scheduler, the pass scheduler and the LSRA resolution pass but always on different data structures. A common implementation using C++ templates would reduce redundancies and increase the reusability of the code. In this context the Boost Graph Library (BGL) (Siek et al., 2001) is worth mentioning.

To some extent this is already done. For instance the loop and dominator analysis are implemented IR independent and are used in the HIR and the LIR part.

Outsource Binary Code Generator

Aycock (2003) pointed out that generating binary code is tedious and error prone. Outsourcing this task into a dedicated library would relieve the CACAO VM community from creating, testing and maintaining this functionality. From the current point of view the following features would be desirable.

Support for jump labels Jumps should be handled via labels. Fall-through jumps (jumps to the following instruction) should not be emitted. It stays open for evaluation if this should be done automatically or in a separate pass.

Support for relocation Many features in CACAO are implemented using loads from a data segment. A binary code library should support this via symbolic labels. This is a generalization of the previous item.

Transparent machine resources Machine resources, for example machine registers or stack-slots, should be usable by the client software. Additionally, virtual registers and virtual stack-slots should be present as an intermediate store for values. These are needed for register allocation.

Instruction meta-information Instructions should provide meta-information, like if it is *commutative*, to the client.

Architecture support Naturally, it should be available for all architectures CACAO supports.

The list above only describes the interface visible to the client. *How* the information in the library is created is another matter. The idea of using architecture description languages (ADLs) is promising and applicable for creating binary tools as shown by Farfeleder et al. (2006). Brandner

et al. (2007) extended this approach and used their structural ADL for generating a tree pattern matching instruction selector. Unfortunately there is no single, broadly excepted ADL, nor are there models for common architectures, at least they are not publicly available. Additionally, ADLs are more focused on custom architectures for special purposes with short time-to-market cycles. For long-living, popular processor families, it seems, that manual implementation is more practical.

Ramsey and Fernandez (1995) proposed the *New Jersey Machine-Code Toolkit*, which targets some of the requirements above. Unfortunately the project seems to be dead, probably due to a missing user base.

There are other stand-alone projects, which are actively maintained, for instance *asmjit*[3] or *jitasm*[4], but they only support a small number of targets. LLVM (Lattner and Adve, 2004) handles the target description using a domain specific language called TableGen (LLVM Documentation, 2013). Although it is not as powerful as "real" ADLs these target descriptions contain most, if not all, information needed for the library described above. Such descriptions exist for all architectures supported by CACAO. TableGen files are used to generate C++ classes used by the LLVM compiler. These classes are interwoven into the LLVM infrastructure. Their direct usability for external projects is limited but the framework allows to write custom TableGen-backends, which use the target description to generate all kind of output files. Creating a stand-alone library based on this TableGen infrastructure seems like a promising approach.

7.1.2 C++11 and the Boost C++ Libraries

The current C++11 standard (ISO, 2012b) contains many language and standard library improvements and is far superior to previous revisions (Meyers, 2005, Item 54, Item 55). Some of the new and many additional features are already part of the Boost C++ libraries (Schling, 2011). Boost is freely available and is supported by most compilers. It provides solutions to many common problems which, can help developers to focus on the real issues. These additional features would improve the readability, maintainability and the performance of the implementation. Therefore the use of C++11 and Boost is strongly recommended.

7.2 Summary

In this work a new second-stage compilation framework for the CACAO VM is described. It is intended for recompilation of frequently executed methods based on the run-time behavior of a program.

The main goal was to create a framework for rapid development of new optimizations. Therefore a new graph-based intermediate representation was introduced. The IR has numerous advantages over the classic form, where the instructions are organized in an ordered list. The SSA form allows pass developers to use simple and efficient algorithms.

To support register allocation and machine code generation, a second, low-level intermediate representation has been created. It is designed to model the target architecture as accurate as pos-

[3] `https://code.google.com/p/asmjit/`
[4] `https://code.google.com/p/jitasm/`

sible without loosing the ability to use target independent algorithms. The high-level compiler pipeline consists of an SSA construction pass, analysis components and three different schedulers. After lowering to the low-level representation, lifetime analysis and register allocation are performed, before the code emitter produces the final binary code.

While the implementation is only in its early stages and there is still a lot of future work to be done, the prototype is already used for implementing and evaluating compiler optimizations.

APPENDIX A

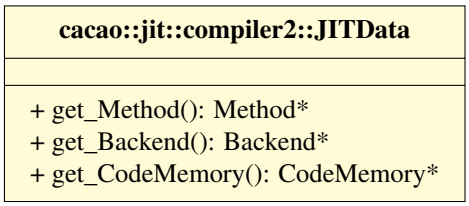
Data Models

A.1 Classes

On the following pages some of the most important classes used in the implementation are depicted. Note that for the sake of better understanding, the models are simplified and only a subset of the members is shown. For example, C++ collections are normally obtained via iterators through a begin()/end() function pair. This is abbreviated with a single public list member, for instance List<Instruction>.

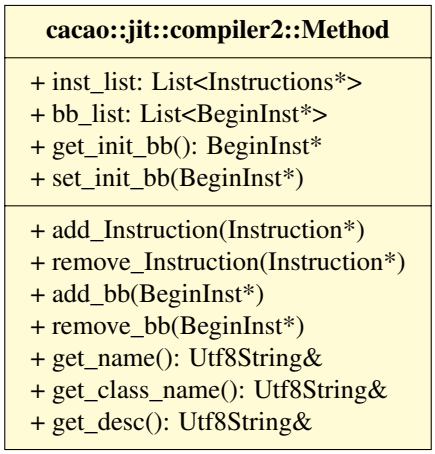
A.1.1 JITData

Class Diagram



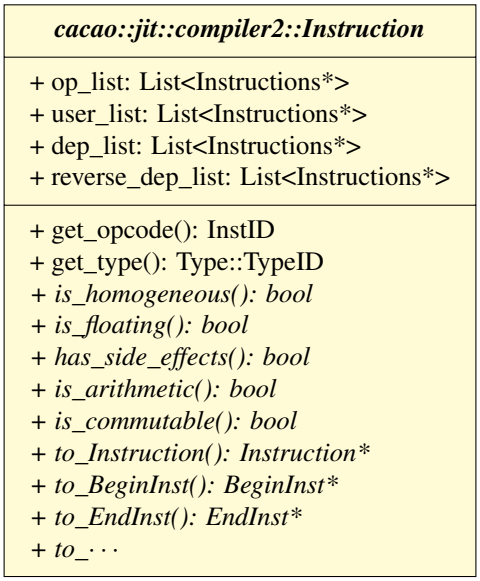
A.1.2 Method

Class Diagram



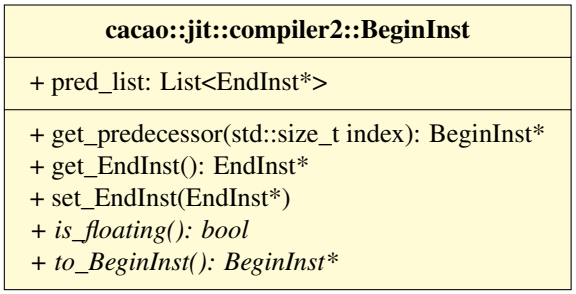
A.1.3 Instruction

Class Diagram



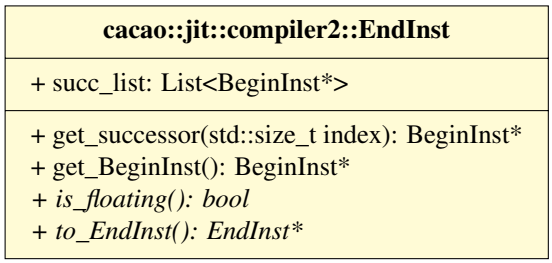
A.1.4 BeginInst

Class Diagram



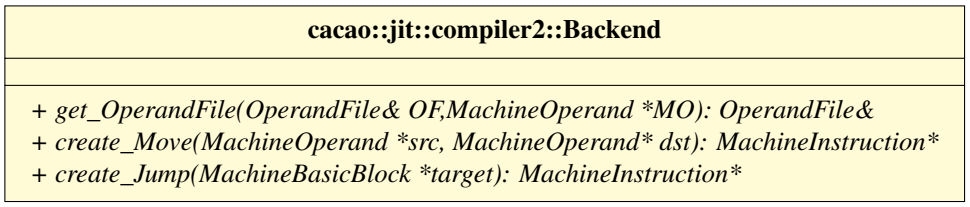
A.1.5 EndInst

Class Diagram



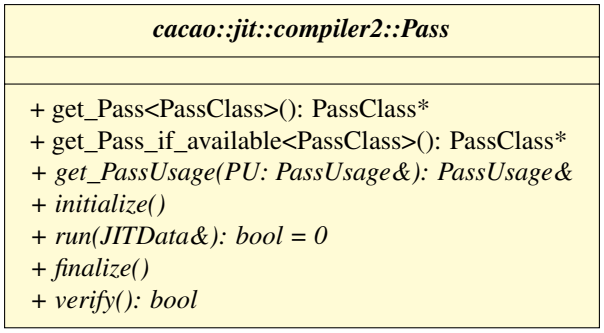
A.1.6 Backend

Class Diagram



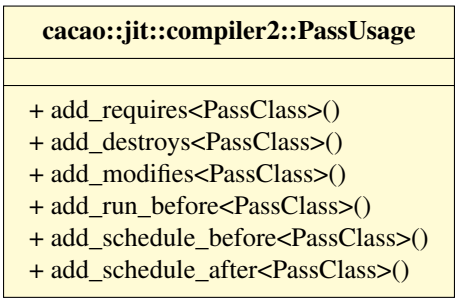
A.1.7 Pass

Class Diagram



A.1.8 PassUsage

Class Diagram



A.1.9 ExamplePass

The ExamplePass is a minimal example for a compiler Pass. It simply prints out all instructions associated with the current Method. Note that the instructions are iterated in an arbitrary order. Both files are part of the source distribution (see Appendix B).

To minimize the required space, the boilerplate comments, which are required by the CACAO coding convention, are omitted.[1]

[1] `http://c1.complang.tuwien.ac.at/cacaowiki/Draft/CodingConventions`

```
#ifndef _JIT_COMPILER2_EXAMPLEPASS
#define _JIT_COMPILER2_EXAMPLEPASS

#include "vm/jit/compiler2/Pass.hpp"

namespace cacao {
namespace jit {
namespace compiler2 {

/**
 * ExamplePass
 *
 * This is an example for a compiler pass that simple prints all
    instructions
 * (in an unrelated order). It can be used as a pass template.
 */
class ExamplePass : public Pass {
public:
        static char ID;
        ExamplePass() : Pass() {}
        virtual bool run(JITData &JD);
        virtual PassUsage& get_PassUsage(PassUsage &PU) const;

        // virtual void initialize();   (optional)
        // virtual void finalize();     (optional)
        // virtual bool verify() const; (optional)
};

} // end namespace compiler2
} // end namespace jit
} // end namespace cacao

#endif /* _JIT_COMPILER2_EXAMPLEPASS */
```

Listing A.1: ExamplePass Header

```
#include "vm/jit/compiler2/ExamplePass.hpp"
#include "vm/jit/compiler2/PassManager.hpp"
#include "vm/jit/compiler2/JITData.hpp"
#include "vm/jit/compiler2/PassUsage.hpp"
#include "vm/jit/compiler2/Method.hpp"
#include "toolbox/logging.hpp"

// define name for debugging (see logging.hpp)
#define DEBUG_NAME "compiler2/ExamplePass"

namespace cacao {
namespace jit {
namespace compiler2 {

bool ExamplePass::run(JITData &JD) {
        Method *M = JD.get_Method();
        // print all instructions (in an arbitrary sequence)
        for (Method::const_iterator i = M->begin(), e = M->end(); i != e;
             ++i) {
                Instruction *I = *i;
                LOG(*I << nl);
        }
        return true;
}

// pass usage
PassUsage& ExamplePass::get_PassUsage(PassUsage &PU) const {
        //PU.add_requires<YyyPass>();
        return PU;
}

// the address of this variable is used to identify the pass
char ExamplePass::ID = 0;

// register pass
static PassRegistery<ExamplePass> X("ExamplePass");

} // end namespace compiler2
} // end namespace jit
} // end namespace cacao
```

Listing A.2: ExamplePass Implementation

APPENDIX B

Source Code Reference

Currently the source code is available to the public in the authors Mercurial[1] repository at:

`https://bitbucket.org/zapster/cacao-compiler2`

Once it is in a stable state, it will be merged into the official repository.

`http://www.cacaojvm.org`

[1] `http://mercurial.selenic.com/`

Bibliography

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811. URL http://dragonbook.stanford.edu/.

B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. *SIGPLAN Not.*, 34(10):314–324, Oct. 1999. ISSN 0362-1340. doi: 10.1145/320385.320418. URL http://doi.acm.org/10.1145/320385.320418.

M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *SIGPLAN Not.*, 46(4):65–83, May 2000. ISSN 0362-1340. doi: 10.1145/1988042.1988048. URL http://doi.acm.org/10.1145/1988042.1988048.

M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840305. URL http://dx.doi.org/10.1109/JPROC.2004.840305.

J. Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL http://doi.acm.org/10.1145/857076.857077.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *SIGPLAN Not.*, 35(5):1–12, May 2000. ISSN 0362-1340. doi: 10.1145/358438.349303. URL http://doi.acm.org/10.1145/358438.349303.

T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16:59–70, 1994.

T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

M. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. In R. Möhring and R. Raman, editors, *Algorithms — ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44180-9. doi: 10.1007/3-540-45749-6_17. URL http://dx.doi.org/10.1007/3-540-45749-6_17.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

H.-J. Boehm. Dynamic Memory Allocation and Garbage Collection. *Comput. Phys.*, 9(3):297–303, May 1995. ISSN 0894-1866. URL `http://dl.acm.org/citation.cfm?id=205931.205941`.

F. Brandner, D. Ebner, and A. Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 13–22, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-826-8. doi: 10.1145/1289881.1289886. URL `http://doi.acm.org/10.1145/1289881.1289886`.

M. Braun, S. Buchwald, and A. Zwinkau. Firm — A Graph-Based Intermediate Representation. Technical Report 35, Karlsruhe Institute of Technology, 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470`.

M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. pages 102–122, 2013. doi: 10.1007/978-3-642-37051-9_6. URL `http://dx.doi.org/10.1007/978-3-642-37051-9_6`.

P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *SIGPLAN Not.*, 24(7):275–284, June 1989. ISSN 0362-1340. doi: 10.1145/74818.74843. URL `http://doi.acm.org/10.1145/74818.74843`.

S. Buchwald, A. Zwinkau, and T. Bersch. SSA-Based Register Allocation with PBQP. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 42–61. Springer Berlin / Heidelberg, 2011. doi: 10.1007/978-3-642-19861-8_4. URL `http://dx.doi.org/10.1007/978-3-642-19861-8_4`. 10.1007/978-3-642-19861-8_4.

G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982. ISSN 0362-1340. doi: 10.1145/872726.806984. URL `http://doi.acm.org/10.1145/872726.806984`.

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer languages*, 6(1):47–57, 1981.

C. Chambers. *The Design and Implementation of the SELF compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

C. Chambers and D. Ungar. Making pure object-oriented languages practical. *SIGPLAN Not.*, 26(11):1–15, Nov. 1991. ISSN 0362-1340. doi: 10.1145/118014.117955. URL http://doi.acm.org/10.1145/118014.117955.

J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. *SIGSOFT Softw. Eng. Notes*, 24(5):21–31, Sept. 1999. ISSN 0163-5948. doi: 10.1145/381788.316171. URL http://doi.acm.org/10.1145/381788.316171.

C. Click and M. Paleczny. A Simple Graph-Based Intermediate Representation. *SIGPLAN Not.*, 30(3):35–49, Mar. 1995. ISSN 0362-1340. doi: 10.1145/202530.202534. URL http://doi.acm.org/10.1145/202530.202534.

C. N. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

A. Coglio. Improving the official specification of Java bytecode verification. *Concurrency and Computation: Practice and Experience*, 15(2):155–179, 2003. ISSN 1532-0634. doi: 10.1002/cpe.714. URL http://dx.doi.org/10.1002/cpe.714.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

P. Dietz and D. Sleator. Two Algorithms for Maintaining Order in a List. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7. doi: 10.1145/28395.28434. URL http://doi.acm.org/10.1145/28395.28434.

G. Duboscq, L. Stadler, D. Simon, C. Wimmer, T. Würthinger, and H. Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Second Asia-Pacific Programming Languages and Compilers Workshop (APPLC 2013)*, Shenzhen, China, Feb. 2013a.

G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. *7th workshop on Virtual Machines and Intermediate Languages*, 2013b. URL https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations.

D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized instruction selection using ssa-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 31–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-104-0. doi: 10.1145/1375657.1375663. URL http://doi.acm.org/10.1145/1375657.1375663.

S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective Compiler Generation by Architecture Description. *SIGPLAN Not.*, 41(7):145–152, June 2006. ISSN 0362-1340. doi: 10.1145/1159974.1134671. URL http://doi.acm.org/10.1145/1159974.1134671.

S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 241–252. IEEE, 2003.

C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, Sept. 1992a. ISSN 1057-4514. doi: 10.1145/151640.151642. URL http://doi.acm.org/10.1145/151640.151642.

C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not.*, 27(4):68–76, Apr. 1992b. ISSN 0362-1340. doi: 10.1145/131080.131089. URL http://doi.acm.org/10.1145/131080.131089.

A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134780. URL http://doi.acm.org/10.1145/1134760.1134780.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542528. URL http://doi.acm.org/10.1145/1542476.1542528.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

C. D. Garret, J. Dean, D. Grove, and C. Chambers. *Measurement and Application of Dynamic Receiver Class Distributions*. Citeseer, 1994.

N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A Lazy Developer Approach: Building a JVM with Third Party Software. In *International Conference on Principles and Practice of Programming In Java (PPPJ 2008)* , Modena, Italy, September 2008.

Graal Project. OpenJDK Community. URL http://openjdk.java.net/projects/graal/.

M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. doi: 10.1145/2500828.2500832. URL http://doi.acm.org/10.1145/2500828.2500832.

S. Hack, D. Grund, and G. Goos. Register Allocation for Programs in SSA-Form. In A. Mycroft and A. Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33050-9. doi: 10.1007/11688839_20. URL http://dx.doi.org/10.1007/11688839_20.

U. Hölzle and D. Ungar. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996. ISSN 0164-0925. doi: 10.1145/233561.233562. URL http://doi.acm.org/10.1145/233561.233562.

U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. *SIGPLAN Not.*, 27(7):32–43, July 1992. ISSN 0362-1340. doi: 10.1145/143103.143114. URL http://doi.acm.org/10.1145/143103.143114.

H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 246–256, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190071.

Intel® 64 and IA-32 Architectures Software Developer's Manual — Volume 3 System Programming Guide. Intel Corporation, March 2013.

ISO. *ISO/IEC 14882:2003: Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2 edition, 2003.

ISO. *ISO/IEC 23271:2012: Information technology – Common Language Infrastructure (CLI)*. International Organization for Standardization, Geneva, Switzerland, 3 edition, 2012a.

ISO. *ISO/IEC 14882:2011: Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, february 2012b.

R. M. Karp. *Reducibility among Combinatorial Problems*. Springer, 1972.

D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

S. R. Kosaraju. Analysis of Structured Programs. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 240–252, New York, NY, USA, 1973. ACM. doi: 10.1145/800125.804055. URL http://doi.acm.org/10.1145/800125.804055.

T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpotTM client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017. URL http://doi.acm.org/10.1145/1369396.1370017.

A. Krall. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 205–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8591-3. URL http://dl.acm.org/citation.cfm?id=522344.825703.

A. Krall and R. Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997. ISSN 1096-9128.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979. ISSN 0164-0925. doi: 10.1145/357062.357071. URL http://doi.acm.org/10.1145/357062.357071.

S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201325772.

G. Lindenmaier. libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Technical Report 2002-5, Sept. 2002. URL http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps.

T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification: Java Se, 7 Ed, 2013. URL http://docs.oracle.com/javase/specs/jvms/se7/html/.

LLVM Documentation. TableGen Fundamentals, 2013. URL http://llvm.org/docs/TableGenFundamentals.html.

S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321334876.

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

J. S. Olesen. Greedy Register Allocation in LLVM 3.0, 2011. URL http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html. LLVM Project Blog.

M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, Sept. 1999. ISSN 0164-0925. doi: 10.1145/330249.330250. URL http://doi.acm.org/10.1145/330249.330250.

M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. *SIGPLAN Not.*, 32(5):109–121, May 1997. ISSN 0362-1340. doi: 10.1145/258916.258926. URL http://doi.acm.org/10.1145/258916.258926.

M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 35–44, 2002. doi: 10.1109/WCRE.2002.1173062.

N. Ramsey and M. F. Fernandez. The New Jersey Machine-Code Toolkit. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 24–24, Berkeley, CA, USA, 1995. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267411.1267435.

F. Rastello, editor. *SSA-based Compiler Design*. Springer, 2013. URL http://www.springer.com/978-1-4419-6201-0. to be released.

B. R. Rau. Levels of Representation of Programs and the Architecture of Universal Host Machines. *SIGMICRO Newsl.*, 9(4):67–79, Nov. 1978. ISSN 1050-916X. URL http://dl.acm.org/citation.cfm?id=1014198.804311.

B. Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN 0982219199, 9780982219195. URL http://www.boost.org.

K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-93798-2. doi: 10.1007/978-3-540-93799-9_2. URL http://dx.doi.org/10.1007/978-3-540-93799-9_2.

J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 2001. ISBN 9780321601612. URL http://www.boost.org/libs/graph/.

L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, VMIL '12, pages 49–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1633-0. doi: 10.1145/2414740.2414750. URL http://doi.acm.org/10.1145/2414740.2414750.

L. Stadler, G. Duboscq, H. Mössenböck, T. Würthinger, and D. Simon. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 9:1–9:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489846. URL http://doi.acm.org/10.1145/2489837.2489846.

M. Starzinger. Exact Garbage Collection for the Cacao Virtual Machine. Master's thesis, Vienna University of Technology, 2011.

E. Steiner. Adaptive Inlining and On-Stack Replacement in a Java Virtual Machine. Master's thesis, Vienna University of Technology, 2007.

E. Steiner, A. Krall, and C. Thalinger. Adaptive Inlining and On-Stack Replacement in the CACAO Virtual Machine. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 221–226, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294356. URL http://doi.acm.org/10.1145/1294325.1294356.

R. E. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9(3): 355 – 365, 1974a. ISSN 0022-0000. doi: 10.1016/S0022-0000(74)80049-8. URL http://www.sciencedirect.com/science/article/pii/S0022000074800498.

R. E. Tarjan. Finding Dominators in Directed Graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974b. doi: 10.1137/0203006. URL http://epubs.siam.org/doi/abs/10.1137/0203006.

R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, Apr. 1975. ISSN 0004-5411. doi: 10.1145/321879.321884. URL http://doi.acm.org/10.1145/321879.321884.

C. Thalinger. Optimizing and Porting the CACAO JVM. Master's thesis, Vienna University of Technology, 2004.

O. Traub, G. Holloway, and M. D. Smith. Quality and Speed in Linear-scan Register Allocation. *SIGPLAN Not.*, 33(5):142–151, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277714. URL http://doi.acm.org/10.1145/277652.277714.

C. A. Vick. SSA-based reduction of operator strength. Master's thesis, Rice University, 1994. URL http://hdl.handle.net/1911/13912.

C. Wimmer and M. Franz. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 170–179, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772979. URL http://doi.acm.org/10.1145/1772954.1772979.

C. Wimmer and H. Mössenböck. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 132–141, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064998. URL http://doi.acm.org/10.1145/1064979.1064998.