FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Instruction Selection for the CACAO VM

MASTERARBEIT

zur Erlangung des akademischen Grades

**Master of Science**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Christian Kühmayer BSc**
Matrikelnummer 0828301

an der Fakultät für Informatik
der Technischen Universität Wien

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

Wien, 14. April 2015

Christian Kühmayer    Andreas Krall

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

# Instruction Selection for the CACAO VM

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Master of Science**

in

**Software Engineering & Internet Computing**

by

**Christian Kühmayer BSc**
Registration Number 0828301

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 14th April, 2015

Christian Kühmayer    Andreas Krall

# Erklärung zur Verfassung der Arbeit

Christian Kühmayer BSc
Steinackergasse 20/5, 1120 Wien

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.

Wien, 14. April 2015

Christian Kühmayer

# Acknowledgements

First, I want to thank my supervisor, Prof. Krall, for his continuous support, for always giving me useful hints and for sharing his extensive knowledge about compiler construction and virtual machines.

I also want to thank the CACAO development team, especially Josef Eisl and Stefan Ring for introducing me to the software, answering countless questions and for helping me with numerous issues.

I want to thank my parents, who always supported my decisions and encouraged me to pursue my ideas. Last but not least, I want to thank my partner, Corina, for her kindness and support.

# Kurzfassung

Befehlsauswahl (Instruction Selection) ist eine wichtige Aufgabe eines Übersetzers. Sie ist für die Auswahl von Maschinenbefehlen verantwortlich, die die Aktionen eines Programmes - oder genauer gesagt dessen Zwischendarstellung - ausführen. Maschinenbefehle sind spezifisch für die jeweilige Zielarchitektur, z.B. unterscheiden sich Maschinenbefehle für x86 und ARM. Daher benötigt diese Aufgabe Wissen über die Zielarchitektur und den verfügbaren Befehlssatz.

Im Gegensatz zu gewöhnlichen Compilern, die Sourcecode zu Maschinencode übersetzen, und Interpretern, die Sourcecode innerhalb eines Aufrufs ausführen, verwenden virtuelle Maschinen einen gemischten Ansatz. Code, der in der Programmiersprache Java verfasst ist wird zu Java Bytecode übersetzt, der von einer virtuellen Maschine ausgeführt werden kann. Diese Darstellung ist vergleichsweise maschinennahe, jedoch nicht spezifisch für eine Zielarchitektur. Eine virtuelle Maschine führt den Bytecode aus, indem sie ihn entweder interpretiert oder über einen JIT-Compiler in Maschinencode übersetzt und diesen ausführt.

Die *CACAO VM* ist eine virtuelle Maschine für Java, die über keinen Interpreter verfügt. Sie verwendet einen schnellen, nicht optimierenden Compiler zur Generierung von Maschinencode für die Programmausführung. Häufig verwendete Teile des Programms können mit einem optimierenden Compiler erneut übersetzt werden um die Ausführung zu beschleunigen.

Diese Arbeit hat das Ziel die Befehlsauswahl des optimierenden Compilers zu verbessern. Sie verwendet Pattern Matching um Maschinenbefehle auszuwählen, deren Ausführung - verglichen mit dem derzeit verwendeten konservativen Ansatz - schneller ist. *lburg* - ein Code-Generator Generator - wird verwendet um den Code der Mustererkennung zu erzeugen der im optimierenden Compiler verwendet wird. Dieser Generator verwendet Spezifikationen in einem Baum-Grammatik Format, die architekturspezifische Muster für die Befehlsauswahl bereitstellen. Die Implementierung des Algorithmus - teilweise generiert - enthält keine Spezifika von Zielplattformen.

Der Code, der von der optimierten Version erzeugt wird, ist bis zu 76% schneller. Die Übersetzungszeit erhöht sich durch das Pattern Matching um etwa 5%, jedoch werden weniger Maschinenbefehle und Operanden emittiert, wodurch sich die Zeit und der Speicherbedarf für die Analyse der Aktivitätsbereiche und die Registerbelegung verringern.

# Abstract

Instruction selection is an important task of a compiler. It is responsible for selecting machine instructions that can perform the actions specified in a program - or to be more precise - a higher level representation of it. Machine instructions are specific to the target architecture, e.g. the machine instructions for x86 and ARM differ. Therefore, this task depends on knowledge about the target architecture and the available instruction set.

In contrast to traditional compilers that translate source code to machine code, and interpreters that execute source code within one invocation, virtual machines take a mixed approach. Code written in the programming language Java is compiled to Java bytecode that can be executed in a virtual machine. This representation is comparatively low level, but not target specific. A virtual machine executes it by either interpreting it, or compiling it just-in-time (JIT) and executing the compiled version.

The *CACAO VM* is a Java Virtual Machine that does not feature an interpreter. It uses a fast baseline compiler to generate machine code for program execution. Frequently used program parts can be recompiled by an optimizing compiler to speed up execution.

This work aims to improve the instruction selection of the optimizing compiler. It applies pattern matching to find machine instructions that execute faster than the ones selected by the currently applied conservative approach. It adapts *lburg* - a code-generator generator - to produce the pattern matching code that is used in the optimizing compiler. This generator uses specifications in tree grammar format that contain target specific patterns for instruction selection. The algorithm implementation - partly generated - is kept platform agnostic.

The code generated by the optimized version runs up to 76% faster. Compile time is increased by about 5% due to pattern matching, but due to less machine instructions and operands emitted lifetime analysis and register allocation are performed faster and require less memory.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

CHAPTER 1

# Introduction

## 1.1 Virtual Machines

Rau identifies 3 kinds of representations of a program [Rau78]:

- High-level representation (HLR), i.e. program source code
- Direct interpretable representation (DIR): A lower representation that is reduced by HLR constructs and should be faster to interpret
- Direct executable representation (DER), i.e. machine code

A traditional compiler translates HLR into DER that can be executed on one specific target platform. Interpreters take a different approach by directly reading HLR and executing it. This eases distribution - the source code can be distributed and will run on every platform that features an interpreter. Executables (DER) are faster than interpreted code, but have to be compiled for every platform.

Virtual Machines make use of direct interpretable representation. The DIR is compiled from HLR (the source code) and can be distributed to the different platforms where a virtual machine executes it. This can either be done by interpreting the DIR or by compiling the DIR to DER and executing it. VMs aim to be faster than interpreters that have to deal with HLR concepts.

The DIR of the Java™ programming language [AGH05] is called Java bytecode [LYBB13]. The *Java HotSpot™ VM*[1] - the most popular Java VM - takes a mixed approach in executing the Java bytecode:

- The bytecode is first interpreted. This way the startup time can be kept low.
- Frequently used parts are compiled to DER and executed natively. This improves the performance of the running program [KWM+08].

[1] www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

## 1.2 The CACAO Java Virtual Machine

The *CACAO Java Virtual Machine*[2] was founded by Andreas Krall and Reinhard Grafl and first provided an implementation for the Alpha Processor [KG97][Kra98]. In contrast to the *Java HotSpot™ VM* which initially interprets bytecode and compiles parts that are used frequently, the *CACAO VM* always compiles the bytecode to native code. This increases startup times compared to interpreting implementations, but does need less resources during execution which allows Java programs to be executed on low powered microcontrollers. It currently supports following target architectures: Alpha, i386, x86_64, ARM, MIPS, Sparc64, PowerPC, PowerPC64, PA-RISC, and s390.

It features a fast non-optimizing baseline compiler to keep startup times short and an optimizing compiler for recompiling frequently used methods to speed up execution. Approaches regarding an interpreter variant have been made [ETK06] but have not been pursued further.

## 1.3 Motivation

In 2013 a replacement for the optimizing compiler was developed [Eis13]. It features several techniques found in modern compilers, e.g. Static Single Assignment (SSA) [EBS$^+$08] [KWM$^+$08] and an extensible pass structure [LA04a][LA04b]. The implementation is not yet complete[3] and compilation times as well as execution times need improvements.

## 1.4 Problem Definition

The new optimizing compiler currently only supports the x86_64 architecture. Lowering of the intermediate representation (IR) to machine instructions is done iteratively: For every high level instruction one or more machine instructions will be generated. As a result, values are copied more often than necessary, the number of instructions is higher and register usage also increases. Unnecessary move operations are eliminated by the register allocator, but this task could often be avoided and compilation time could be reduced. Further, many processors feature complex instructions that combine some calculations into one step, e.g. Multiply-Add. The iterative approach cannot utilize such commands as multiply and add are different instructions in the intermediate representation and lowered separately.

## 1.5 Aim of the Work

The lowering task should be improved by applying a pattern matching algorithm. The algorithm should be able to handle directed acyclic graphs (DAGs). Alternatively graph splitting has to be done to apply tree pattern matching. The implementation should be

[2]www.cacaojvm.org

[3]Development fork is maintained at BitBucket until it is ready for production

target independent, target specific content should ideally only be supplied by means of patterns used by the algorithm.

The effect on compilation time should be kept low. Hence, applying heuristics may be favorable to more complex algorithms.

## 1.6 Methodical Approach

Instruction selection is done by all compilers and is a well researched topic. If possible, an existing approach should be reused. Following questions have to be answered in evaluating existing solutions:

- Is it fast enough to work in a JIT compiler?
- Can it be integrated into the optimizing compiler? Important things to consider are: Is the intermediate representation suitable as input or does it need to be adapted? Can the result be processed with the existing compiler infrastructure? Solutions that are not based on C/C++ might also be hard to integrate.
- Does it also provide patterns for the target architectures? This would help supporting additional targets.
- Is its license compatible with GPL[4] used by *CACAO VM*?

If an existing solution can be reused, it has to be adapted and integrated into the optimizing compiler. If not, it is necessary to develop a specialized solution. In this case the theoretical work behind an existing solution might be reused.

## 1.7 Structure of the Work

The remainder of the work is structured as follows: Chapter 2 gives an overview of the state of the art of instruction selection as well as related topics such as different intermediate representations and related compiler passes.

In chapter 3 the status quo of the optimizing compiler is discussed and a decision is made on the approach used. It then shows how the existing architecture has to be changed to implement the solution. Afterwards the implementation will be discussed.

Chapter 4 shows which optimizations can be done with the implemented solution. It outlines the different kinds of optimizations and gives some examples of actual optimizations done for the supported target architecture.

The results of the optimizations are evaluated in chapter 5. It provides benchmarks and compares the results of the compiler with and without pattern matching. The remainder of the work critically reflects on the work done (chapter 6) and provides a conclusion (chapter 7).

[4]www.gnu.org/copyleft/gpl.html

CHAPTER 2

# State of the Art

## 2.1 Compiler Architecture

The architecture of a compiler is usually split into two parts. The front end, which is responsible for parsing the code and building and optimizing a high level intermediate representation. And the backend that generates executable code from the intermediate representation. The main focus of this work is instruction selection, which is the first pass in a compiler backend. The following sections deal with the intermediate representation and the backend passes.

## 2.2 Static Single Assignment

Static single assignment (SSA) intermediate representation was developed by IBM researchers [RWZ88] [AWZ88] [CFR+91] and is used by several important compilers and virtual machines (e.g. *GCC* [Nov03], *LLVM* [LA04a], *Java Hot Spot VM* [KWM+08]).

In SSA a variable is only defined once. Cases where variables are redefined can be transformed into SSA representation by replacing every redefinition with a definition of a new variable. In cases where a variable is defined by multiple values (e.g. depending on the branch taken) so called PHI operations are introduced that merge the values. The SSA form simplifies the representation as reassignments are prevented and therefore it is well suited for performing optimizations. Listing 2.1 shows a small example of a conditional variable assignment using SSA representation.

## 2.3 Graph based SSA IR

Click and Paleczny [CP95] proposed a graph based intermediate representation that is in SSA form. It completely omits variables and instead uses nodes for operations which define a value. Using a value is expressed by def-use edges that provide a link between

**Algorithm 2.1:** Example SSA Representation. Source: [CFR+91]

```
1 if $P$ then
2     $V_1 \leftarrow 4$
3 else
4     $V_2 \leftarrow 6$
5 end
6 $V_3 \leftarrow \phi(V_1, V_2)$
```

the defining operation and the operation that requires the value as input operand. The example in figure 2.1 shows a SSA graph of a loop incrementing a variable.

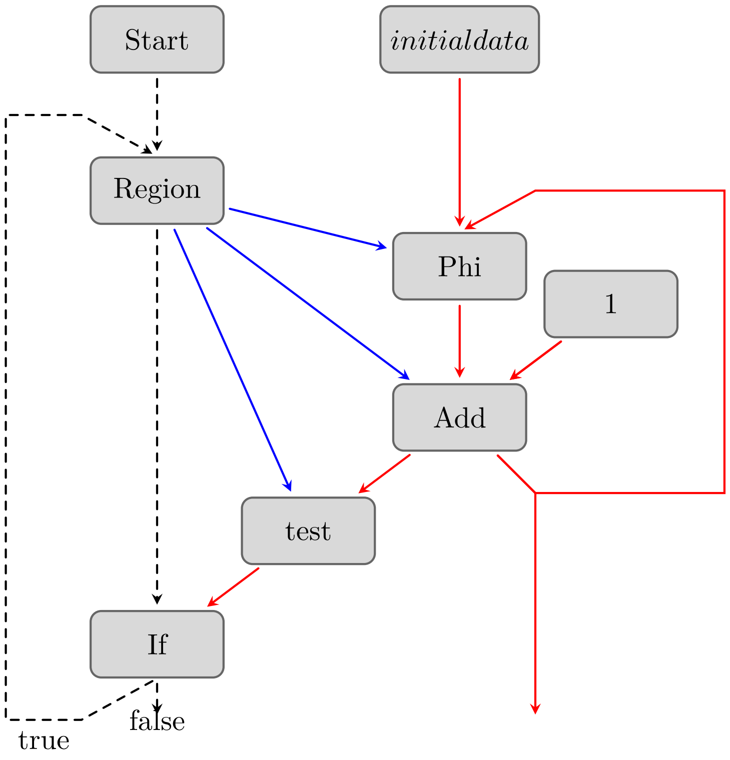


Figure 2.1: SSA graph example adopted from [CP95]. Control flow is shown as dashed black edges, scheduling dependencies are denoted by blue edges, data dependencies are shown in red.

## 2.4 Instruction Selection

Instruction selection is the task of choosing machine instructions that perform the actions modeled in the intermediate representation. It is the first task that requires knowledge of the target architecture. Depending on the kind of intermediate representation and compile time constraints there are several algorithms to choose from.

### 2.4.1 Tree Pattern Matching

Intermediate representations that are centered around statements in the program code usually contain trees. Pattern matching using a bottom up approach has proven to be an efficient way of instruction selection and is used by *Twig* [AGT89] which generates a table driven matcher, *BEG* [ESL89] and *burg* [FHP92b].

*burg* uses BURS (Bottom-Up Rewrite System) theory and works with cost augmented tree grammars. Dynamic programming is already done at compile-compile time which requires the cost values to be constants. *burg* finds an optimal tree cover in linear time.

*iburg* [FHP92b] works with the same grammars but its implementation is shorter and simpler and moves dynamic programming to compile time. This causes a penalty in runtime compared to burg but allows dynamic cost calculation.

*lcc* is a compiler that incorporates *lburg* [FH95] which is an adapted version of *iburg*. Its tree pattern grammar accepts arbitrary strings instead of constant integer values for the cost attributes. This can be used to specify a cost calculation that is inserted into the generated matcher code.

An example *lburg* tree grammar can be seen in Listing 2.1. The rule in line 16 specifies an ADD operation with a constant value. The helper function `constant_encodable(a)` returns a cost of 1, if the constant is encodable as immediate, *MAX_COST* otherwise. Loading a constant (line 14) and using the standard ADD (line 15) has a total cost of 2. If the constant can be encoded the rule in line 16 will account for a cost value of 1 and will be used in favor of the standard case.

Although dynamic cost calculation increases compile time, it might be favorable for being able to consider type information that is not directly represented in the intermediate representation.

Tree pattern matching can also be applied to directed acyclic graphs. The DAG will be split into trees which are then matched with following assumptions:

- Nodes referenced from within the tree that are not part of the tree any more are assumed to be values located in a register. (e.g. 2.1 line 13 will be matched instead of whatever is actually present in the IR).
- The result of a tree root node must always be stored to a register.

Tree pattern matching provides optimal instruction selection for trees, but these are only local optima. Overall the instruction selection might not be optimal for the DAG.

#### BURG Algorithm

All *burg* derivates have a two-pass structure. First, a labeling function traverses the tree bottom-up, left-to-right and determines a minimum cost cover. Second, the tree is reduced by traversing it top down and performing the actions specified for the determined

```
%{
// includes
...
// declarations
int constant_encodable(Node* a);
}%
%start stm
%term ADD = 1
%term CONST = 2
%term REG = 3
%term LOAD = 4
%%
reg: REG                "/*code␣to␣emit*/" 1
reg: LOAD(CONST)        "/*code␣to␣emit*/" 1
reg: ADD(reg, reg)      "/*code␣to␣emit*/" 1
reg: ADD(reg, CONST)    "/*code␣to␣emit*/"
  constant_encodable(a)
%%
// helper functions
int constant_encodable(Node* a){
    if (...) return 1;
    return MAX_COST;
}
```

Listing 2.1: Simplified lburg specification example

rules. This pass is usually implemented by the user whereas the labeling pass is generated based on the grammar.

The labeling pass determines all rules and cost values for each node in the tree. There might exist several rules that match the terminal node, differing in the left hand side (lhs) nonterminal and in the parameters. For each left hand side nonterminal a data structure is created within the state structure of the node that records the chosen rule and its cost value. If a matching rule derives from the same lhs nonterminal and has smaller cost, the rule and cost value will be overwritten. There can also be chain rules, i.e. nonterminals that are derived to other nonterminals. In this case, the cost is only forwarded. Traversing the tree every node is annotated with the best rules for each lhs nonterminal and the cost values. Upon reaching the root node, the rule with the minimum cost is chosen which implicitly defines the rules chosen for every other node in the tree. The generated algorithm can be seen in procedure 2.2 (slightly simplified), for a detailed description see [FHP92b] and [FHP92a]. After the minimum cost cover is found, the reduce pass will traverse the tree and take the actions specified for the rules.

**Procedure 2.2:** burmLabel(node)

```
switch $node.terminal$ do
    // one case-statement generated for every terminal symbol
    case $terminalA$ ...;
    case $terminalB$
        // recursive labeling (omitted, if child node not present)
        burmLabel($node.leftchild$);
        burmLabel($node.rightchild$);
        /* one if-block generated for each child nodes combination.  for
           leaf nodes if condition and child nodes are omitted          */
        if $(node.leftchild.terminal = terminalX) \wedge (...)$ then
            // add all child costs and rule cost
            $cost \leftarrow$ State[$node.leftchild$].$cost$[$param\_nt$] $+ ... +$ $rule\_cost$
            if $cost <$ State[$node$].$cost$[$lhs\_nt$] then
                State[$node$].$cost$[$lhs\_nt$] $\leftarrow cost$
                State[$node$].$rule$[$lhs\_nt$] $\leftarrow rule$
            end
        end
    end
endsw
```

### 2.4.2 Extending Tree Pattern Matching to DAGs

Ertl shows how tree pattern matching can be extended to DAGs [Ert99]. If a value is used multiple times the intermediate representation is a DAG. As described in 2.4.1 tree pattern matching requires the DAG to be split. The calculation result is forced into a register which is then accessed multiple times. Revisiting the example in listing 2.1 two ADD operations referencing the same constant value node as operand would not be able to utilize the rule in line 16.

This problem typically occurs with address arithmetic. Many machine instructions not only support encoding addresses but also address calculations using registers, multiplications and offsets. Hence the whole tree calculating the address can often be encoded without emitting any machine code. As memory access often happens more than once - e.g. when fetching a value and storing the results of a calculation back to the same address - encoding address calculations cannot be done efficiently with standard tree pattern matching.

Ertl extends a tree pattern matching algorithm to parse DAGs. It keeps track of already visited parts and if a subgraph is derived several times using the same nonterminal, the subgraph can be shared, i.e. optimizing rules can be chosen. If the nonterminal is not the same the subgraph might be reduced in different ways which leads to code duplication. In this case, splitting the graph and storing the result of the subgraph to a register is more efficient.

### 2.4.3 DAG Pattern Matching

Digital signal processors often feature irregular instructions that are highly efficient but hard to utilize as the IR result is a graph. Even today, performance critical parts often remain hand written assembler code for that reason. [EKS03] and [EBS$^+$08] solve this problem using a SSA DAG intermediate representation for instruction selection. As finding a minimum cost cover for a DAG is NP-hard they translate the SSA graph into a Partitioned Boolean Quadratic Problem (PBQB) and use a heuristic solver to find a solution. This provides near optimal results and keeps compilations times reasonable, but it is not suited for JIT compilation.

### 2.4.4 LLVM

The compiler framework *LLVM* [LLV] uses a DAG-to-DAG pattern matching technique for instruction selection. The intermediate representation of *LLVM* is used to create a graph, that at first only consists of target independent nodes. These nodes - or combinations of nodes - will be replaced by target specific nodes according to patterns which are specified in so called *TableGen*-Specifications. After all nodes are suited for the target the instructions in the graph will be scheduled and a list of machine instructions is generated. According to Koes et al. [KG08] *LLVM*'s default instruction selection uses a maximal munch algorithm [AP03]. This algorithm matches a node and subsequent tree child nodes greedily. They propose another algorithm that does near optimal instruction selection on DAGs in linear time. The *LLVM* approach might be of interest in a few other aspects too:

- The *TableGen*-Specifications contain the specifications for the target machines, i.e. patterns for the DAG-to-DAG matching, opcodes, register information, instruction set extensions etc.
- They are available for all targets supported by *LLVM*. This means, that all the important architectures are specified in this format.
- *TableGen* is a program that reads a target specification and builds an in-memory model of it. It is used within *LLVM* to generate target specific code which is included in the compiler. It is possible to write custom backends for it to customize the output regarding content and format.

Designing the pattern matching and the backend in a way that makes use of *TableGen*-Specifications might dramatically ease supporting other targets with the optimizing compiler.

## 2.5 Scheduling vs Register Allocation

Instruction selection does not necessarily order the machine instructions. Further, the output uses so called virtual registers, i.e. placeholders for actual machine registers. The task of scheduling and allocating registers is left to subsequent passes.

There are several approaches about the order of the passes [GH88]:

- Prepass scheduling: Scheduling is done before register allocation. The scheduler arranges the instructions to minimize wait times and does not take care of register usage. This approach leads to more spills, i.e. code that is inserted to temporarily move content from registers to memory.
- Twopass scheduling: Scheduling is done before and after register allocation to optimize the schedule after spill code has been inserted.
- Postpass scheduling: Register allocation is done first, which minimizes spills. This restricts the number of possible schedules and might introduce wait times.
- Register aware prepass scheduling algorithms: The scheduler keeps track of the number of required registers and can switch scheduling strategies to avoid spilling.

The register aware scheduling algorithms proposed by the authors outperformed the other approaches. Regarding the simple approaches, prepass scheduling seems to be favorable.

### 2.5.1 Scheduling

As mentioned above, scheduling tries to arrange instructions in a way that minimizes wait times. Multiple computation units and pipeline architectures allow scheduling of non conflicting instructions in an interleaved way to utilize unused resources. Warren defines a list scheduling algorithm for systems with multiple computation units that arranges the instructions and avoids wait times [War90]. Provided a dependency graph, it first determines which instructions are eligible for scheduling (i.e. have no dependencies). It will choose the instruction farthest away from the graph root. The def-use edges' weight specifies the wait time required for the preceding instruction to provide a result. Following the edges from the root to the eligible nodes these values can be summed up. The node with the highest value will be scheduled and removed from the dependency graph. This is repeated until no instruction is left. The scheduling approach by Gibbons and Muchnick solves the problem for pipelined architectures in a similar way [GM86].

### 2.5.2 Register Allocation

Register allocation can be expressed as a graph coloring problem which is NP-hard. Chaitin [Cha82] proposed heuristics to solve this graph problem and spill registers to memory if necessary. Some improvements [BCKT89] and other graph coloring heuristic solutions [CH90] have been proposed, but similar to section 2.4.3 heuristics for NP-hard problems are usually not fast enough for JIT compilers.

Poletto and Sarkar [PS99] describe an algorithm called linear scan register allocation (LSRA) which scans the live range of registers in one linear pass - hence the name - and is considerably faster than any allocator using graph coloring. As such it is an ideal candidate for JIT compilers. It is optimized and implemented by Wimmer and Mössenböck in the *Java HotSpot™ VM* [WM05]. Wimmer and Franz further show how LSRA can be used directly on an IR in SSA form [WF10]. Omitting the creation of another representation saves compilation time and memory and the characteristics of SSA form help the algorithm to produce slightly better machine code.

## 2.6 Java HotSpot™ VM

The *Java HotSpot™ VM* features an interpreter and two optimizing compilers. The client compiler is described in [KWM+08] and features a high level SSA IR, low level machine code IR, LSRA, etc. However the article does not mention how instruction selection is done. The highly optimizing server compiler uses BURS tree pattern matching for instruction selection [PVC01].

## 2.7 Graal VM

The *Graal VM* is a modification of the *Java HotSpot™ VM* replacing the optimizing compilers with the Graal Compiler. Graal is written in Java and its intermediate representation is an SSA graph [DSW+13] that is similar to the one proposed by Click et al. [CP95]. The IR makes heavy use of annotations to declare the dependencies in the graph representation.

Speculative optimization is also considered in the design of the IR [DWS+13]. Unlikely program branches (e.g. exception handling) are pruned and not part of the optimized version. If the assumptions do not hold, deoptimization is done. The VM state is reconstructed to execute the pruned branch via the interpreter. The IR supports this by inserting special nodes, e.g. Guard-nodes for null checks or FrameState-nodes that note the bytecode index for the interpreter to start when deoptimization happens.

One of the most important optimization techniques is inlining as it avoids function calling overhead. Inlining is often done at an early stage, e.g. on bytecode level or during generation of the IR. The *Graal VM* performs late inlining and uses a graph cache to speed up compilation [SDMW12]. Before building the IR for a function to be inlined, the graph cache is looked up. If the graph IR is already available in the cache it can be directly used. This is particularly rewarding for functions that are inlined frequently.

CHAPTER 3

# Implementation

## 3.1 Status Quo

This part will give an overview on the currently implemented features of the *CACAO VM* optimizing compiler and provide a foundation for the implementation decisions made in section 3.2. A detailed description can be found in [Eis13].

The compiler is organized in passes. Each pass can specify dependencies to other passes that are required to precede it, e.g. to provide necessary input. The passes will be scheduled according to these dependencies provided a schedule is found. The early passes are responsible for constructing the SSA graph and for performing high level optimizations such as constant propagation, constant folding, etc. Later passes are responsible for scheduling, instruction selection and register allocation. Figure 3.1 shows the later passes and their dependencies relevant for this work.

The optimizing compiler uses a graph based SSA IR similar to the one proposed by Click and Paleczny [CP95] (see section 2.3) and implements a scheduling concept proposed by Click in his PhD thesis [Cli95]. Instruction selection is usually done in a basic block scope, but the HIR does not provide these boundaries. Click's scheduling concept is responsible for determining basic blocks, scheduling the basic blocks and assigning the high level instructions to the basic blocks. Within basic blocks the optimizing compiler uses list scheduling to order the high level instructions.

Afterwards instruction selection and scheduling is done within the MachineInstructionSchedulingPass. The instruction selection is currently done iteratively and cannot combine HIR instructions for optimization purposes. Improving this part of the compiler is the main subject of this work. The low level intermediate representation (LIR) instruction ordering follows the list scheduling order of the HIR instructions, i.e. no further scheduling is done, as it is already valid, but scheduling of LIR instructions could improve execution time.

After handling loops in LIR, the lifetime analysis will be done. Its analysis is used as input for the LSRA (see section 2.5.2). Afterwards the machine code is created by using

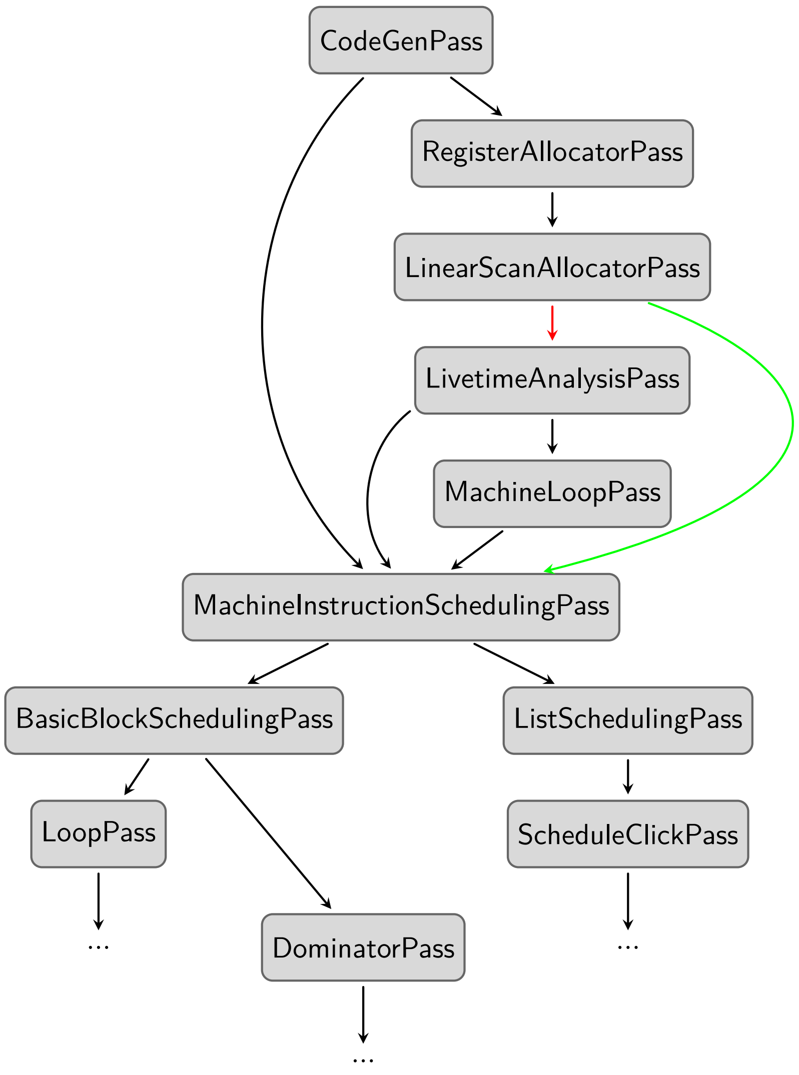


Figure 3.1: Late compiler passes (adopted from [Eis13]). Black arrows denote *strong* dependencies. Green edges show *modifications* and red edges show *destroys* postconditions

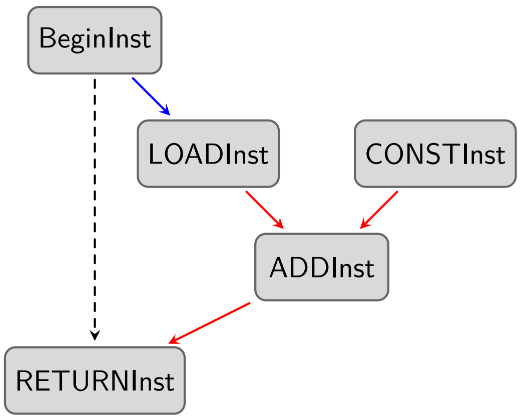


Figure 3.2: HIR graph of a function that adds a constant value to a parameter and returns the result. Blue arrows indicate scheduling dependencies, red arrows show data dependencies. The dashed arrow associates the BEGINInst and ENDInst (in this case of subtype RETURNInst). No CFG edges are shown as the function only consists of one basic block.

opcodes and encoding operands such as registers, memory locations and immediates and the result is emitted into memory.

### 3.1.1 A closer look at the basic block and instruction selection

Figure 3.2 shows the HIR of a simple function adding two integers and returning the result. One of the integers is provided via parameter, the other value is a constant. After the list scheduling pass instruction selection is performed by processing every high level instruction separately. All high level instructions are subclasses of the Instruction class and instruction selection is done using the visitor pattern. The result is a low level intermediate representation (LIR) that models the machine instructions. Lowering[1] involves creating several objects for the low level intermediate representation:

**MachineInstruction:** Subtypes of this class represent machine instructions and contain information like opcodes and encoding functions.

**Mov-Instructions:** Subclasses of MachineInstruction frequently used for x86_64. In two-operand machine code one of the source operands will also be used as destination. To prevent overwriting values that might be required somewhere else, one operand will first be copied into another location which will also be used to store the result after the machine instructions is executed.

[1]As instruction selection creates a low level IR it is also referred to as lowering

**MachineOperands:** Represent values that are defined or used by machine instructions. Important subclasses are:

- VirtualRegister: Used as placeholder, will be resolved to a machine register by the register allocator.
- NativeRegister: Specifies an actual machine register for instructions that require specific registers.
- Immediate: Constant values that can be moved to a register or encoded into an instruction.

As can be seen in figure 3.3, the result is far from optimal. First, move operations are introduced that are not needed. They are later removed by the register allocator, but it would speed up compilation if they are not created in the first place. Second, the constant value is represented as a separate high level instruction and is therefore lowered on its own, which results in the constant value being moved to a virtual register which will be used when the add instruction is lowered. All this can be avoided, if the constant can be encoded into the add instruction as immediate value. However it is not possible to utilize this feature with this approach.

## 3.2 Implementation Decisions

**Pattern matching on DAGs** is not an option for the JIT nature of *CACAO VM*. Even solutions that apply heuristics (see section 2.4.3) are not suited runtime performance wise.

**LLVM DAG-to-DAG matching** was particularly interesting due to the *TableGen*-Specifications as they would help supporting additional target architectures. It has not been evaluated regarding runtime as there were other issues discovered:

- As noted in section 2.4.4 *LLVM* builds a separate DAG for the matching task. This should be avoided to keep runtime consumption and memory usage down. But this would imply adaptations to the HIR of the *CACAO VM*'s optimizing compiler as otherwise the patterns that already exist within the *LLVM* project would not match.
- The *TableGen*-Specifications are not always complete: Things that are hard to specify in a declarative fashion are often omitted and instead handled in hand written C++ code. E.g. x86_64 opcodes are not specified but code emission uses hard coded hex values and encodes the operands with it.
- Constraints for applying patterns often rely on hand written code: E.g. for ARM architectures not all constants can be encoded as immediate value. *LLVM* as well uses C++ functions for such cases.

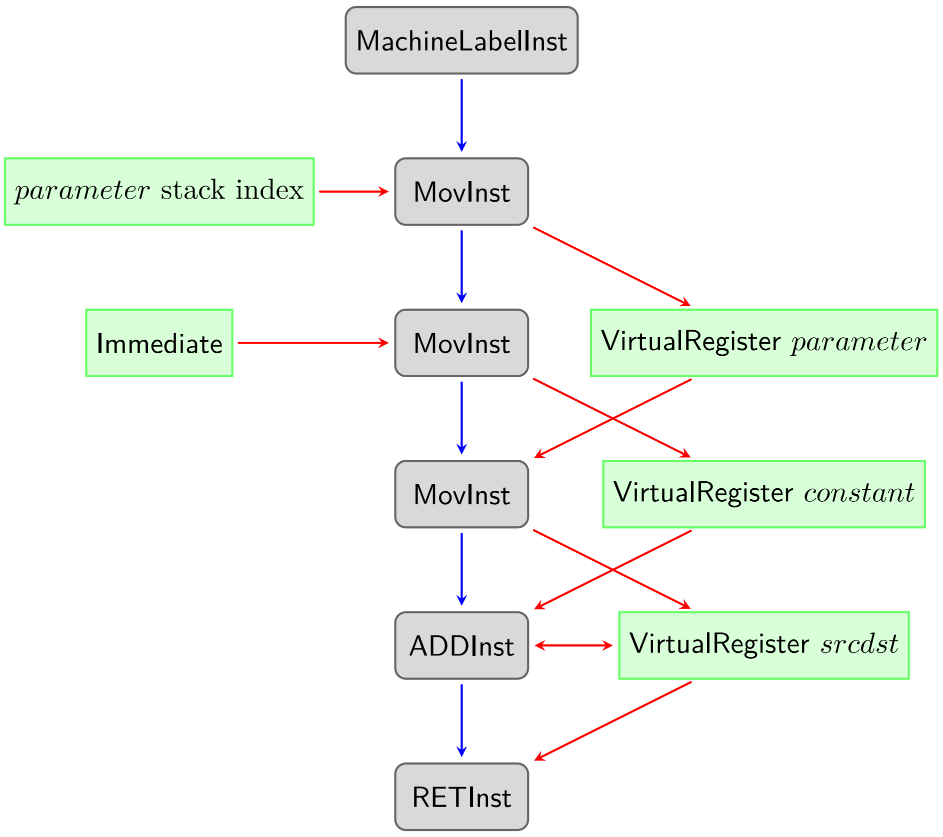


Figure 3.3: LIR Result of Iterative Lowering. Operands are shown in green, machine instructions are grey. Blue arrows denote the ordering of machine instructions. Red arrows show data dependencies (ADDInst reads and writes the same virtual register).

- The *TableGen*-Specifications are highly coupled with *LLVM* code. C++ functions, classes, enums etc. are directly referenced by the specifications. Using the specifications would require significant effort to either port *LLVM* code into *CACAO VM* - which might be a licensing issue - or writing equivalent code.

Because of these drawbacks reusing *LLVM* pattern matching is not an option. When extending the architecture support of the optimizing compiler it might still provide some options. If the architecture to be supported is specified with opcodes and operands writing a *TableGen* backend and generating code for the code emission task might be a viable solution. This is not in the scope of this work but should be considered for future tasks.

**Tree pattern matching** is therefore the obvious choice. It is faster and does not impose the JIT compiler with a high runtime penalty. And it can also be used for DAGs by splitting them.

The extension proposed by Ertl (see section 2.4.2) is considered for future optimizations. The optimizing compiler's capabilities for non-static content are limited at the moment. Memory access is only done for static variables, stack content, and arrays and only primitive types are supported. The first two cases have only minor optimization potential and array access is handled by separate HIR instructions which can incorporate the optimization without the need for pattern matching. Once the optimizing compiler supports non-static content address arithmetic calculations will happen quite often and this optimization should be considered.

### 3.2.1 Starting Point

*lburg* is chosen as basis for this work as it enables dynamic cost calculation without further adaptations. The Instruction subclasses used in the HIR are not subclassed regarding data type; this information is only available via member variable/function. Subsequently the grammar cannot incorporate type information which should affect decisions. With dynamic costs the type information can be accessed and the cost value can be calculated accordingly to influence rule selection.

The *lburg*-generated code requires adaptations before it can be included in the context of the optimizing compiler. The essential algorithm emitted should not need any customization.

As the license of *lcc*[2] (the compiler suite incorporating *lburg*) conflicts with the GPL used by *CACAO VM* the generator code is kept in a separate repository[3].

## 3.3 Changes to the Pass Structure

Instruction selection is done in the MachineInstructionSchedulingPass. It requires two preceding passes: BasicBlockSchedulingPass which schedules the basic blocks, and ListSchedulingPass which schedules the HIR Instructions within a basic block. The input to the MachineInstructionSchedulingPass is in both cases a schedule, i.e. a list of pointers to the basic blocks / instructions. The lowering task will generate machine instructions with respect to these schedules.

Pattern matching is done with basic block scope and creates machine instructions, so it naturally fits into the MachineInstructionSchedulingPass. But there is no need for a schedule within the basic block before that task, so the ListSchedulingPass is not needed. At the moment pattern matching should only be optional and list scheduling can not be removed. Instead both options should be selectable via *configure* options.

To support this strategy, list scheduling is removed as a separate pass and integrated into the MachineInstructionSchedulingPass. This pass inherits the dependencies of the ListSchedulingPass and depending on the configuration either list scheduling and iterative lowering or pattern matching will be performed.

[2]License information and source code at github.com/drh/lcc
[3]bitbucket.org/kuetsch/cacao-patternmatching

## 3.4 Grammar Design

The *lburg* input grammar is already shown in listing 2.1. The generated code is included into a Matcher class to retain the object structure used in the optimizing compiler. This has a few effects on the grammar input file. As the generated file will never be compiled on its own the includes and declarations in the top part can be left empty and will be provided by the including file. The bottom part which usually features helper functions will only be used for methods that are architecture specific. Commonly used methods are defined in Matcher. The string token in every rule that usually contains the assembler code is used to specify a rule identifier that is used in an enum definition. This way the rule specific content can be accessed by name rather than a numeric value.

### 3.4.1 Excluded Instructions

*lburg* supports instructions with at most two input operands. Some HIR instructions exceed that limits, e.g. PHIInst, or INVOKESTATICInst as function calls allow an arbitrary number of parameters. Adding support would cause significant effort and performance improvements are unlikely. Hence, these instructions will not be handled by pattern matching and they will not appear in any grammar rule. The implications on adapting the generator and designing the Matcher are discussed in the subsequent sections.

### 3.4.2 Terminals

Every Instruction in the HIR has an `opcode` member of enum-type InstID. The same names and integer values are used in the grammar to define the associated terminal symbols. The specification only supports terminals with fixed arity, but RETURNInst does not follow this approach. It is used as ending statement in a function and the operand it takes represents the return value. But it can also be used for `void` functions without a return value. In contrast to the excluded instructions in section 3.4.1 including RETURNInst in pattern matching is desired. One way to cope with this is to create separate HIR instructions for these cases. The current matcher implementation solves it by assuming the operand to be present and providing a stub if it is missing (see section 3.8.4).

### 3.4.3 Basic Rules

As HIR instructions are not subclassed for different data types, the rules cannot distinguish between them. But the HIR itself is already consistent regarding data type. E.g. adding different data types is solved by inserting a CASTInst for one operand and the ADDInst itself never gets operands with different data types as input.

This has some interesting implications on the nonterminals and basic rules that do not perform any optimization. As nonterminal definition and usage always ensure compatible data types no distinction has to be made. In fact, a nonterminal represents a def-use edge in the SSA graph that also has no type information attached. As a result the grammar can be modeled with only one nonterminal. This nonterminal is trivially used as start terminal in the grammar and for all operands. An excerpt can be seen in listing 3.1.

```
%{
%}
%start stm
...
%term ADDInstID = 10
%term SUBInstID = 11
%term MULInstID = 12
%term DIVInstID = 13
%term REMInstID = 14
%term SHLInstID = 15
%term USHRInstID = 16
%term ANDInstID = 17
%term ORInstID = 18
%term XORInstID = 19
%term CMPInstID = 20
%term CONSTInstID = 21
...
%%
stm: ADDInstID(stm,stm) "ADDInstID" 1
stm: SUBInstID(stm,stm) "SUBInstID" 1
stm: MULInstID(stm,stm) "MULInstID" 1
stm: DIVInstID(stm,stm) "DIVInstID" 1
stm: REMInstID(stm,stm) "REMInstID" 1
stm: SHLInstID "SHLInstID" 1
stm: USHRInstID "USHRInstID" 1
stm: ANDInstID(stm,stm) "ANDInstID" 1
stm: ORInstID(stm,stm) "ORInstID" 1
stm: XORInstID(stm,stm) "XORInstID" 1
stm: CMPInstID(stm,stm) "CMPInstID" 1
stm: CONSTInstID "CONSTInstID" 1
...
%%
...
```

Listing 3.1: Basic Grammar

These basic rules are the equivalent to iterative lowering. When matched only one HIR instruction will be processed. They provide a minimum implementation without optimizing rules. Every HIR instruction is matched by one rule and the Matcher forwards the instruction selection task to the existing LoweringVisitor. The identifier string token is identical to the terminal and allows easy identification of basic rules by checking if the identifier value is in the InstId value range.

## 3.5 Generated Grammar Content

Handling the large number of Instruction subclasses can be a tedious task. Hence Eisl developed a small *Python* script that generates some include code. It is used e.g. for Instruction IDs, and for generating the `visit()` method declarations for the lowering visitor. The script requires a csv file as input that lists all the instruction class names and generates the output files accordingly.

This python script and the accompanying csv file are extended to generate code and grammar content for the matching process. The csv file is extended by a column denoting the arity of the instruction. If the column is empty and no arity is given, the instruction is not suitable for pattern matching. These instructions will be listed in the file ExcludedNodes.inc and used by the Matcher that checks for excluded instructions (see section 3.4.1). The instructions that define an arity are used to generate the basic grammar content, i.e. the terminal definitions and the basic rules as shown in listing 3.1.

The optimizing grammar not shown in the example (denoted by `...` around the separator in line 32) is specified in a separate file. Whereas the basic grammar is only available once and identical for all target architectures, the optimizing grammar is target specific and located in each architecture subfolder.

## 3.6 Workflow

Both the *Python* script and the generator have to be called manually before building the *CACAO VM*. As the script also creates input for the generator it has to be executed first.

The workflow is slightly complicated due to the fact that the generator is located in a separate repository. To simplify working with the generator, its makefile executes all steps with one invocation. It assumes that the *CACAO VM* repository is cloned to the same folder as the generator repository and that its folder is named `cacao-compiler2`. It first compiles the generator, then it executes the *Python* script located in the VM repository that generates include content and the basic grammar. For each architecture the basic grammar and the optimizing grammar part are then concatenated to a complete grammar file. Then the generator is called for every architecture with this complete grammar as input. It generates the Grammar.inc file which is included by the Matcher. All generated (intermediate) files except the ones generated by the *Python* script are located in the respective architecture subdirectories.

As the pattern matching generator is not part of the *CACAO VM* repository and build process, the generated files are committed to the repository. Otherwise building the VM would fail. Editing the grammar requires the pattern matching repository, but as long as other tasks are handled, the developer is not required to clone the repository.

## 3.7 Generator Adaptations

*lburg* is only modified with respect to the generated output. The generated content is split into several parts that are surrounded by `#ifdef` macros. Each part can be included on its own at the appropriate location in the Matcher code. The generated content is only slightly adapted. An additional enum RuleId is generated based on the string token in the grammar. This is used to identify the rules and to execute the corresponding code. The method definitions are prefixed with the class name as the generated function declarations are included within the Matcher class definition. Logging and aborting execution is also adapted to use the *CACAO VM* provided functions and macros. Manual memory allocation is replaced by `new` and `shared_ptr<>`. Other minor changes related to a specific topic are discussed in the respective chapters.

## 3.8 Matcher

The Matcher class incorporates the generated code and is instantiated within the MachineInstructionSchedulingPass for every basic block. This section covers the implementation as well as the applied algorithms.

### 3.8.1 Interfaces

The Matcher constructor accepts 3 arguments:

- Pointer to GlobalSchedule: This is provided by the ScheduleClickPass (that was preceding ListSchedulingPass) which is responsible to assign the Instructions to basic blocks.
- Pointer to BeginInst: The begin instruction is used as marker in the GlobalSchedule to find the instructions associated with the basic block.
- Reference to LoweringVisitor: This class is used by the visitor pattern for the iterative instruction selection. It is target specific (e.g. a typedef for the actual X86_64LoweringVisitor) and has `visit(...)` methods for all HIR instructions. It is used for lowering of single instructions and extended with the method `lowerComplex(...)` that handles lowering of instructions that match optimizing rules.

For every basic block a Matcher object is instantiated and the method `run()` is called which performs all necessary tasks for the instruction selection.

### 3.8.2 Determining Trees

The GlobalSchedule is first iterated with respect to the basic block to collect all instructions that are contained. Pointers to the found instructions are kept in a set. This allows easy checks if an instruction is part of the basic block which would otherwise require iterating over the global schedule again for searching it. The PHIInsts will be stored into another set as they are not part of the matching and always have to be scheduled directly after a BeginInst.

The set of instructions within the basic block is then iterated and DAG splitting is performed. An Instruction can be the root of a subtree for following reasons:

- The node is excluded from matching and must not be part of a tree: Upon matching the trees the root nodes are again checked and excluded nodes are skipped. Instead of pattern matching the node is passed on to the existing LoweringVisitor.
- The node has multiple users. This is the classic DAG splitting case.
- The node has no users. This might occur if calculated values are discarded.
- The node's user is outside of the basic block: Pattern matching is only done with basic block scope so the node will serve as a tree root within the basic block.
- The node's user is an excluded node: As excluded nodes are lowered separately, they assume that the operand value is available in a VirtualRegister.

The matcher keeps track of the subtrees as well as the dependencies (and reverse dependencies) between them to order the output properly. Finding and handling dependencies is discussed in section 3.8.5.

### 3.8.3 Labeling Trees

The labeling pass requires some type and macro definitions that can be seen in listing 3.2. The `typedef` in line 1 defines the HIR instructions as tree nodes. Line 2 makes another `typedef` for the state structure. Although it is generated by *lburg* it could be exchanged easily. The `#define`s provide methods to access the tree or the state structures associated with the nodes. Line 4 and 7 are simple mappings, but accessing the children of a subtree in a DAG is more complicated and handled in a separate method. For every tree determined (see section 3.8.2) the labeling function is called. The generated algorithm is discussed in section 2.4.1.

```
1 typedef Instruction* NODEPTR_TYPE;
2 typedef struct burm_state STATE_TYPE;
3
4 #define OP_LABEL(p) ((p)->get_opcode())
5 #define LEFT_CHILD(p) (getOperand(p, 0))
6 #define RIGHT_CHILD(p) (getOperand(p, 1))
7 #define STATE_LABEL(p) (state_labels[p])
```

Listing 3.2: Type and Macro Definitions

### 3.8.4 Providing Stubs and Determining Dependencies

The matcher avoids creating separate tree data structures for the labeling pass. It directly uses the HIR that is not in tree form. The method `getOperand()` is responsible for presenting it to the labeling pass like a tree by providing stub nodes. It also finds dependencies between trees and is therefore one of the more complex hand written parts. Both the stubs and the dependencies are stored in the Matcher object. The stubs will be directly used by the labeling pass whereas the dependency information is only relevant during the reduce phase.

In the simplest case, this method only returns the operand that is modeled in the HIR. This is the default action when operating inside the tree. Handling the *borders* of a tree that results from DAG splitting requires some special handling. A tree correctly ends at its leaves, i.e. nodes without operands. DAG splitting results in nodes that have operands that are not part of the tree (see figure 3.4). The tree pattern matching is not allowed to incorporate the child node for matching, but the tree is incomplete without a leaf. In this case the requested operand is replaced by a stub for the labeling pass.

The class NoInst is a subclass of the HIR class Instruction. This node never appears in the HIR graph itself and is used exclusively for stubs. When accessing a child node via `getOperand()` the method checks if a stub is already existing or necessary and returns the corresponding instance. Using the `get_operand()` method of the Instruction class always returns the actual HIR node. The algorithm is shown in function 3.1.

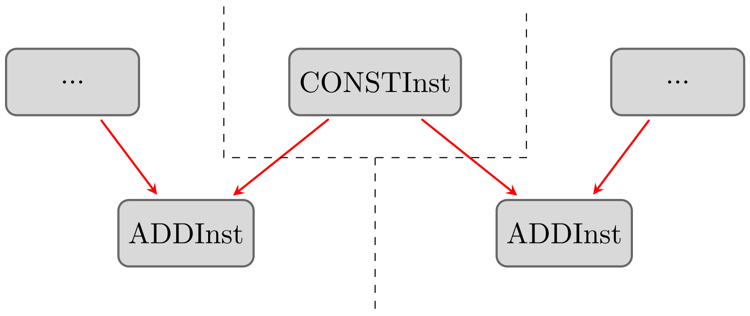


Figure 3.4: DAG Splitting: Both ADDInst nodes use the same CONSTInst node. The DAG is split up in 3 trees: CONSTInst will be a separate tree with only the root node. The ADDInsts in the other two trees will be matched with NoInst stubs replacing the constant definition.

The `getOperand()` method detects following cases and provides stub nodes:

- If the requested child node is missing a stub will be provided. This is only the case for RETURNInst of type TypeID:VoidTypeID.
- If the requested child node is the root of another tree in the same basic block (as determined by 3.8.2) a stub is provided. This is also true for excluded nodes, as they are treated like tree root nodes. Further the data dependency to the other tree is tracked.

- If the requested child node is not part of the basic block it is also stubbed. In this case, no dependency needs to be tracked.

**Dependency Tracking:** The trees are always identified by their root nodes. If a dependency is detected between trees it is entered into the dependency and reverse-dependency data structure. This requires knowledge of the root node of the currently handled tree. To prevent traversing the tree bottom-up until it is found the labeling function is extended by a second parameter that always passes the root of the current tree. For convenience, this information is also stored into the state structure.

**Function 3.1:** getOperand(parentInst, index)

```
// use stub, if it already exists
if $\exists$Stubs[$parentInst$][$index$] then
    return Stubs[$parentInst$][$index$]
end
// get actual operand
$operand \leftarrow parentInst.operands[index]$
/* create stub if operand does not exist, or is a tree root, or is an
   excluded node, or is not part of the basic block                    */
if ($\nexists operand$) $\lor$ ($operand \in$ Roots) $\lor$ ($operand \in$ ExcludedNodes) $\lor$ ($operand \notin$
BasicBlockInstructions) then
    // create and set stub
    $stub \leftarrow new\ NoInst()$
    Stubs[$parentInst$][$index$] $\leftarrow stub$
    // for roots (and excluded nodes) track dependencies
    if ($operand \in$ Roots) $\lor$ ($operand \in$ ExcludedNodes) then
        $treeRoot \leftarrow$ State[$operand$]$.root$
        Dependencies[$treeRoot$] $\leftarrow operand$
        ReverseDependencies[$operand$] $\leftarrow treeRoot$
    end
    // return stub
    return $stub$
end
// return actual operand
return $operand$
```

## 3.8.5 Scheduling

The first instructions scheduled in a basic block are always the BeginInst followed by all PHIInst. They are always lowered before the actual scheduling within the basic block starts (see procedure 3.2 line 1-4).

On a basic block level the trees are scheduled as a whole one after the other with respect to dependencies. The algorithm considers trees to be scheduled if they have no dependencies to other trees and will choose the tree that has the most reverse dependencies (i.e. other trees depending on it). The tree with the highest count is scheduled and removed from the dependencies and reverse dependencies data structures. This is repeated until no trees are left for scheduling. The only constraint is, that the tree containing the EndInst (e.g. RETURNInst) is scheduled at last. See procedure 3.2 starting from line 5 for this part.

Scheduling of the specific tree is done as shown in procedure 3.3. If the tree root is an excluded node, it consists only of this one node and will be lowered using the default lowering visitor method. Otherwise the procedure is called in recursive fashion for all child rules and afterwards the procedure to lowering the parent itself is called.

Lowering a single rule is shown in procedure 3.4. If the rule identifier is an instruction identifier, it denotes the lowering of a single instruction (see section 3.4.3). In this case the existing LoweringVisitor approach is used, provided that the instruction is not a stub in which case no lowering has to be performed at all. Otherwise, a rule with more than just one instruction is the subject, and complex lowering is performed. The complex lowering gets the root instruction and the rule identifier as parameters and creates the LIR representation accordingly.

**Procedure 3.2:** scheduleTrees

```
   // lower beginInst and all PHI instructions
1  lower(BEGINInstruction)
2  foreach $phi \in$ PHIInstructions do
3      lower($phi$)
4  end
   // loop until no tree is left
5  while unscheduledTrees $\neq \emptyset$ do
6      foreach $tree \in$ unscheduledTrees do
7          if Dependencies[$tree$] $\neq \emptyset$ then
               // skip tree if it depends on another tree
8              continue
9          end
10         if ($tree \supset ENDInst$) $\wedge$ (unscheduledTrees $\setminus$ $tree \neq \emptyset$) then
               /* skip tree if it contains ENDInst and other trees still have
                  to be scheduled                                            */
11             continue
12         end
13         if ReverseDependencies[$tree$].$size$ > ReverseDependencies[candidate].$size$
           then
               /* if more trees are depending on the tree, it becomes the new
                  candidate                                                  */
14             candidate $\leftarrow$ $tree$
15         end
16     end
       // lower the tree
17     lowerTree(candidate.$rootInst$, $1$)
18 end
```

**Procedure 3.3:** lowerTree($rootInst, nonTerm$)

```
1 if $rootInst \in \mathsf{ExcludedNodes}$ then
      // default lowering for excluded instructions
2     lower($rootInst$)
3 end
  // get state struct with results of labeling pass
4 $instState \leftarrow \mathsf{State}[rootInst]$
  // get number of matched rule via helper function
5 $rulenumber \leftarrow$ getRule($instState$, $nonTerm$)
6 foreach $(childRule, childInst)$ of $rootInst$ do
      /* recursively lower subtrees using its root instruction and
         nonterminal                                              */
7     lowerTree($childInst, \mathsf{NonTerminals}[rulenumber][childRule]$)
8 end
  // lower tree using its root instruction and rule id
9 lowerRule($rootInst$, $rulenumber$)
```

**Procedure 3.4:** lowerRule($rootInst, ruleId$)

```
1 if $ruleId \in \mathsf{InstId}$ then
2     if $rootInst.opcode = \mathsf{NoInstID}$ then
          // stub.  no lowering required
3     else
          // default lowering of a single instruction
4         lower($rootInst$)
5     end
6 else
      // lowering of multiple instructions at once
7     lowerComplex ($rootInst, ruleId$)
8 end
```

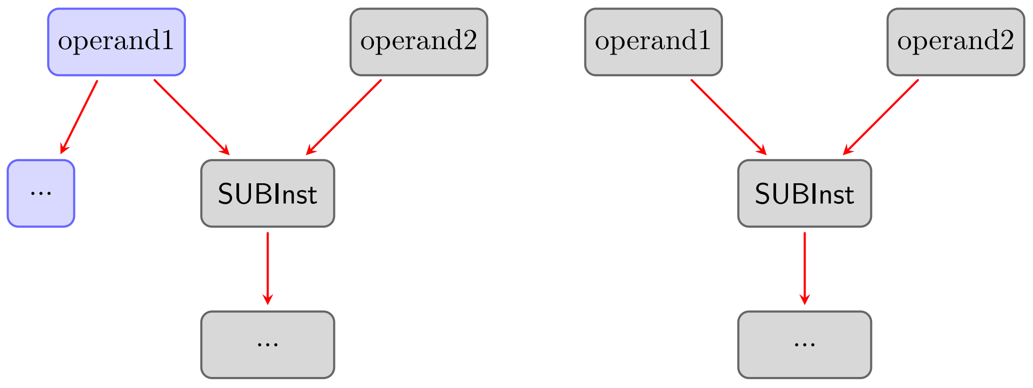
CHAPTER 4

# Optimizations

## 4.1 Avoiding Copies

As shown in figure 3.3 the iterative lowering process creates many unnecessary copies. As two address instructions use the first operand as destination this operand has to be copied first to avoid overwriting the original value which might be used somewhere else. The register allocator will remove unnecessary copy operations, but avoiding them where possible would speed up the register allocation.

Checking the usage of every operand of an instruction during lowering is quite elaborate. Withing a tree the values are never accessed more than once and this information can be stored during the labeling pass. The state structure is extended by a `copyOperands` boolean that is set to `false` for all nodes that only use operands that are part of the tree. For all nodes that use values defined outside of the tree it is



(a) Copy required (b) Copy can be avoided

Figure 4.1: Copies can be avoided for values that are not reused. Nodes in blue denote instructions that are not part of the matched tree.

set to `true`. Figure 4.1a shows a SUBInst where the first operand is defined outside of the matched tree. In this case copying the operand is required. In figure 4.1b the first operand is part of the matched tree. As can be seen, the SUBInst is the only user of `operand1`. Hence, creating a copy can be avoided and overwriting the source operand is safe.

### 4.1.1 Extending the Generator

This optimization requires some modifications to the generator. For every tree node that is labeled the generated code is extended by one line that checks if any of the operands is of type NoInst. If a stub is detected the node is not part of the tree (as in figure 4.1a) and the boolean value is set in the handled node to indicate the usage of *foreign* values.

### 4.1.2 Extending the Visitor Pattern

As described in section 3.4.3 the basic rules reuse the visitor pattern for lowering single high level instructions. Every Instruction subclass features a `void accept ( LoweringVisitor& v )` method that calls the LoweringVisitor's `void visit ( Instruction* I )` method with its actual Instruction subtype. Both methods are extended by a `bool copyOperands` parameter. Its value is true for all cases outside of pattern matching (e.g. lowering BeginInst and PHIInst) as well as for calls when pattern matching is deactivated. With pattern matching the value is determined by the boolean value defined in section 4.1.1.

The LoweringVisitor is extended by a method to setup the operands for the classic two-operand instructions. If no copy is required and the first operand is of type VirtualRegister it will be directly used as destination. The method also considers exchanging operand order for commutable instructions (e.g. ADDInst) and switches operands if the second operand is a VirtualRegister.

### 4.1.3 Possible Improvements

This optimization is only done for the most basic case. Instructions that require more specific handling of operands (e.g. DIVInst or REMInst) are not considered but might also benefit from applying it.

It is currently only used for basic rules. As complex rules lower multiple HIR instructions at once a single boolean parameter is not enough. The `lowerComplex()` method would be required to access the state structure created during the labeling pass to access the `copyOperands` information for every node.

Another improvement would be possible by storing the information for every operand of an instruction. The current implementation creates a copy of the first operand even if only the second operand is not part of the tree. However, this approach has to take care of different arities and might lead to more complexity in the visitor pattern and the passing of the parameter(s).

## 4.2 Encoding Immediates

Many instructions allow encoding of immediate values into the machine code. As constant values are a separate HIR instruction this optimization can not be done by iterative lowering. The patterns used for this optimization can be seen in listing 4.1.

```
stm: ADDInstID(stm, CONSTInstID) "AddRegImm"
  encodeDiscreteCost(a, a->get_operand(1))
stm: SUBInstID(stm, CONSTInstID) "SubRegImm"
  encodeDiscreteCost(a, a->get_operand(1))
stm: MULInstID(stm, CONSTInstID) "MulRegImm"
  encodeDiscreteCost(a, a->get_operand(1))
```

Listing 4.1: Encoding Immediates

For all three instructions, a constant can be encoded to avoid moves and the creation of VirtualRegisters. The terminal can be directly specified as parameter, it is not necessary to provide another nonterminal. The cost calculation is only shown in a shortened way: The encoding is currently only done for discrete values, hence the TypeID of the instruction has to be ByteTypeID, IntTypeID or LongTypeID. Further, encoding is limited to 4 byte, so another check has to be done, if the value of CONSTInst fits into 4 byte. If all conditions are met, the cost value will be 1 whereas it will be a high value (constant $MAX\_COST$) otherwise. The cost calculation in this example is *misused* for constraints. The grammar does not provide a way to specify constraints that have to be met for using the rules. But they can be enforced by the cost calculation that produces very high cost values. If the value can be encoded and the cost is 1 it will be used in favor of two basic rules, that account for a total sum of 2 (see listing 3.1 line 19 and 30). If the constant cannot be encoded the cost value of $MAX\_COST$ is too high and these two rules with total cost of 2 will be used instead.

Although ADDInst and MULInst are commutative no rules with exchanged operand order are specified. Operations with constants are handled separately during creation of the HIR and always place the CONSTInst as second operand for commutative instructions[1]. If the order is changed by optimizations or by changes done to the SSAGraphConstructionPass the optimizing rules have to be created accordingly.

Non-commutative instructions with a constant as first operand are not suitable for encoding immediates. For SUBInst this would require the second parameter to be negated first and afterwards an add instruction with encoded immediate value could be issued. In the end no reduction of machine code or register usage can be achieved this way.

## 4.3 Combining HIR Instructions

Previous optimizations only combined instructions with CONSTInst which is a separate HIR instruction but not a real one in machine code. This section shows how actual

[1]Operations with two constant operands should be eliminated by constant folding optimization before instruction selection takes place.

instructions can be combined to utilize machine instructions that perform multiple operations within one step.

### 4.3.1 The LEA Instruction

The Intel architecture often allows one of the operands to be located in memory. The address of the operand does not need to be calculated beforehand. Instead the most common address arithmetic calculations are supported by the machine instructions and can be encoded.

The LEA instruction's only action is to calculate this address arithmetic and to store the resulting address into a register (see [Int] for instruction details). It therefore performs less tasks than MOV that uses the address and loads / stores a value from / to the memory location or other commands that use the value in the specified location. As it is not accessing the memory it can be used for arbitrary calculations.

```
lea dst, [base + index * scale + displacement]
// dst, base, index: registers
// scale: 2,4,8
// displacement: max. 4 byte immediate value
```

Listing 4.2: Address Calculation

Listing 4.2 shows how the address calculation is structured. *base* usually contains the begin of an array in memory, *index* selects the n-th element and is multiplied by the *scale* factor that takes care of the element size. An arbitrary, signed *displacement* value can further be added (which also enables subtractions). The only constraint for this value is that it has to fit into 4 bytes. All values are optional and can be omitted which makes it quite versatile.

### 4.3.2 Implementing LEA Optimizations

Figure 4.2 shows an example of a HIR graph that can be optimized by using a LEA machine instruction. As *base* and *index* are required to be registers, they do not need to be part of the matched tree. Several other possible forms exist that can be optimized with the LEA instruction. The order and operands of the instructions might be different and as all values are optional they and the respective operations might be missing. All these permutations have to be specified as patterns to do this optimization where possible. Listing 4.3 shows some patterns of HIR instructions that can be calculated with a single LEA instruction. The example omits the identifier token and the cost calculation for clarity. The cost calculation is also used for constraints, similar to section 4.2. It assures that *scale* has the value 2, 4 or 8 and that *displacement* fits into 4 byte.

### 4.3.3 Further Optimizations

The work only highlights some examples how pattern matching can be used. The examples only handle a few instructions regarding discrete values. There may be lots of

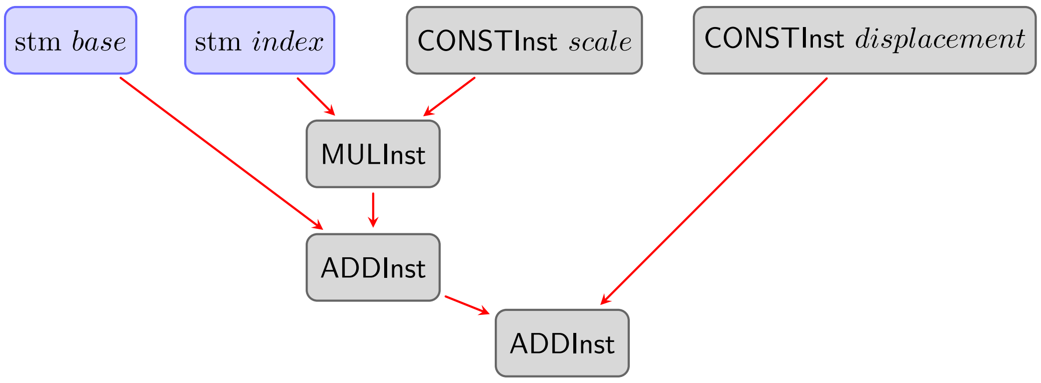


Figure 4.2: A HIR example that can be solved with LEA Optimizations. All operands are optional. *base* and *index* have to be registers and do not need to be part of the matched tree. *scale* has to be 2,4 or 8. *displacement* has to fit into 4 bytes.

```
stm: ADDInstID(ADDInstID(stm, stm), CONSTInstID)
stm: ADDInstID(stm, ADDInstID(stm, CONSTInstID))
stm: ADDInstID(stm, MULInstID(stm, CONSTInstID))
stm: ADDInstID(MULInstID(stm, CONSTInstID), stm)
stm: ADDInstID(ADDInstID(stm, MULInstID(stm, CONSTInstID)),
  CONSTInstID)
stm: ADDInstID(stm, ADDInstID(MULInstID(stm, CONSTInstID),
  CONSTInstID))
...
```

Listing 4.3: LEA Patterns

other optimizations for discrete and floating point instructions that profit from pattern matching.

CHAPTER 5

# Evaluation

## 5.1 Methodology

The compiler performance is evaluated regarding compilation time, allocated memory, register allocator results and code size. The execution of the resulting machine code is only evaluated regarding execution time. The compiler is limited to static content and arrays, that are not affected by the implemented optimizations, so memory consumption can be left out of the comparison. Execution times will be averaged from 30 executions to eliminate influences by the system. For the same reason, the benchmarks are executed after a reboot with no other applications running.

### 5.1.1 Building the *CACAO VM* Executables

Two builds of the *CACAO VM* are created from the same SW version. One build is done for iterative lowering, the other build configures pattern matching[1]. Both builds have debugging disabled[2], optimizations enabled[3] and activated timing[4], statistics[5], and logging[6] options for compiler evaluation.

### 5.1.2 Test Environment

The test system is comprised of a 2.66 GHz Intel® Core™ 2 Duo processor, 8 GB DDR3 RAM (1067 MHz bus speed) and a Samsung 830 Series SSD on an SATA II port. The system runs OS X Yosemite 10.10.2.

[1]-DPATTERN_MATCHING compiler flag; will be a configure option in the future

[2]–disable-debug

[3]–enable-optimizations

[4]–enable-rt-timing

[5]–enable-statistics

[6]–enable-logging

## 5.2 Benchmarks

There are several benchmarks available for measuring integer performance, e.g. Dhrystone [Wei84] or the more modern CoreMark [GO]. The algorithms would have to be ported to Java as they are written in C or other compile-only languages. Fhourstones [Tro] is an exception as it is available in Java as well. DaCapo [DaC] and SPECjvm [Sta] are Java benchmark suites that perform various real life tasks such as XML transformation, vector graphics calculation etc. However, these benchmarks cannot be used as the optimizing compiler is not mature enough for running fully fledged benchmarks.

### 5.2.1 Status Quo

Eisl uses micro-benchmarks for evaluating the performance of the optimizing compiler [Eis13]. Evaluation is done by compiling single methods and writing the object code to file. This object code is linked and called for evaluation.

An easier solution is already available and being used for the *JUnit* test suite. They compile and execute methods via a Java Native Interface (JNI)[7] method. In contrast to the above mentioned benchmarks everything can be coded in Java, but only the one specified method is compiled by the optimizing compiler. Everything else is compiled by the baseline compiler. Each test case is comprised of two methods: The `@Test` annotated *JUnit* test case, and a method that is the test candidate. Each test case compiles and executes a test candidate twice: first using the baseline compiler, then with the optimizing compiler. The criteria for a succeeding test case is return value equality between the two methods.

### 5.2.2 Adaptations

**Measuring Execution Time:** The micro benchmarks implemented in this work reuse the *JUnit* test infrastructure that is already available. The test framework is slightly adapted for the benchmark usage. The existing JNI method is split up into two methods, one responsible for compiling, the other one for executing the compiled code. The Java call to the executing method is surrounded by code for time measurement. It uses `System.nanoTime()` and logs the execution time of every call.

**Measuring Compiler Performance:** Measuring compiler performance is already implemented and can be done with the respective *configure* options (see section 5.1.1). Due to the changes to the pass structure (see section 3.3) MachineInstructionSchedulingPass incorporates both available solutions. Depending on the compiler switch the pass performs either HIR list scheduling and iterative lowering or pattern matching. The evaluation compares the results of MachineInstructionSchedulingPass and the subsequent passes.

[7]docs.oracle.com/javase/7/docs/technotes/guides/jni/

### 5.2.3 Benchmark Methods

Due to the limitations of the micro-benchmarks, the test methods are usually small and finished within several $\mu s$. Function calling overhead and system influences often account for most of the time measured and can easily cause values that are several times higher. To outrule these influences long running benchmarks are desired. The implemented benchmarks use for-loops to accomplish this. An example can be seen in listing 5.1. Note that *base* and *index* are modified within the for loop to prevent code motion optimizations from moving the calculation out of the for loop. All benchmarks will be called with a large count value to accomplish a runtime of $100ms$ to $300ms$.

```
static int base_indexScale__Displacement(int count, int base,
 int index) {
    int a = 0;
    for (int i=0; i <= count; i++){
        base = base+i;
        index = index+i;
        a = (base + (index * 2)) + 3;
    }
    return a;
}
```

Listing 5.1: Benchmark for LEA Optimization

Following benchmarks have been implemented to test the optimizations:

- Add Immediate: Performs several ADDInst with CONSTInst as operand to evaluate performance gains of encoding immediate values.
- Mul Immediate: Performs several MULInst with CONSTInst as operand. Unlike ADDInst encoding immediate values causes a 3-operand instruction.
- Several LEA Benchmarks: To test LEA optimizations, the implemented patterns have to be reflected in the test code. All tested patterns can be seen in table 5.1, benchmark LEA5 is already shown in listing 5.1.

**Existing Unit Test Methods:** The following test methods are included in some comparisons: boyerMoore, sqrt, permut, matAdd, matMult, and matTrans. They give insights on how the changes affect more complex methods that do not profit that much from the optimization patterns.

| Benchmark | Pattern |
|---|---|
| LEA1 | $(base + index) + displacement$ |
| LEA2 | $base + (index + displacement)$ |
| LEA3 | $base + (index * scale)$ |
| LEA4 | $(index * scale) + base$ |
| LEA5 | $(base + (index * scale)) + displacement$ |
| LEA6 | $base + ((index * scale) + displacement)$ |

Table 5.1: LEA Benchmarks and Associated Patterns

## 5.3 Results

The newly created benchmarks target the implemented optimizations and are therefore not conclusive about overall performance. As a counterpart the existing benchmarks are compared, but their execution time is rather short. As soon as more complex benchmarks are possible the optimizations should be reevaluated.

### 5.3.1 Execution Time

Execution time comparison can be seen in table 5.2 and figure 5.1. Overall a significant improvement can be observed. Targeted benchmarks are 43.37% faster on average with LEA5 improving by 76.09%. MulImm did not show any runtime differences. This could be caused by the optimization requiring an additional register compared to the 2-operand MUL instruction.

The existing unit tests have also been evaluated regarding execution time (see table 5.3). Their execution time is shorter as they are not looped. The effect of the optimizations is much smaller, as the code does not make heavy use of them. Overall an improvement of 3.62% can be observed with a peak improvement of 22.18% for matAdd.

| Benchmark | Iterative Lowering | Pattern Matching | Comparison |
|---|---|---|---|
| | $ms$ | $ms$ | $1 - IL/PM$ in % |
| AddImm | 257.67 | 192.97 | -33.53 |
| MulImm | 228.83 | 229.21 | -0.17 |
| LEA1 | 152.80 | 116.40 | -31.28 |
| LEA2 | 166.86 | 119.61 | -39.51 |
| LEA3 | 176.48 | 116.99 | -50.85 |
| LEA4 | 178.55 | 126.44 | -41.21 |
| LEA5 | 206.53 | 117.28 | -76.09 |
| LEA6 | 201.20 | 115.22 | -74.62 |

Table 5.2: Comparison of execution time for targeted benchmarks

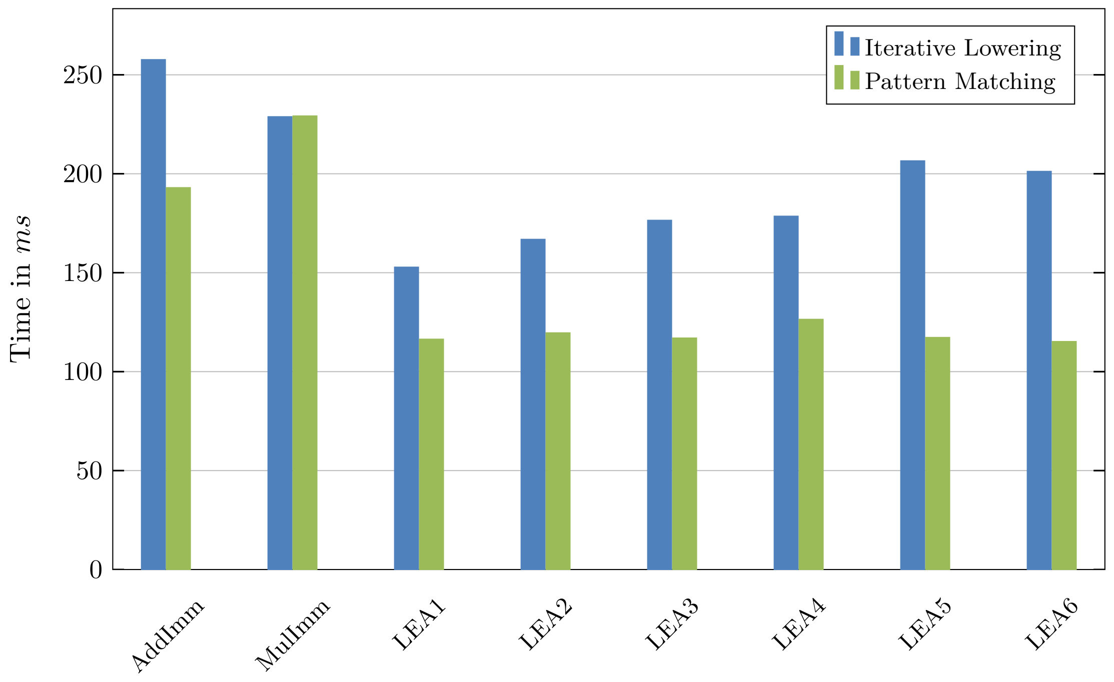


Figure 5.1: Comparison of execution time between Iterative Lowering and Pattern Matching.

| Benchmark | Iterative Lowering | Pattern Matching | Comparison |
|---|---|---|---|
| | $\mu s$ | $\mu s$ | $1 - IL/PM$ in % |
| boyerMoore | 11.53 | 11.20 | -2.98 |
| sqrt | 6.76 | 6.60 | -2.53 |
| permut | 12.77 | 13.13 | 2.79 |
| matAdd | 10.47 | 8.57 | -22.18 |
| matMult | 14.17 | 14.70 | 3.63 |
| matTrans | 7570.77 | 7534.03 | -0.49 |

Table 5.3: Comparison of execution time for existing unit test methods

### 5.3.2 Compilation Time

As can be seen in table 5.2 and figure 5.2 the time needed for compiling the functions mostly increased. The only notable exception is AddImm that compiled 4.08% faster. On the other end of the range LEA2 shows an increase in compile time of 11.65%. Overall compile time increased by 5.30%.

The newly created benchmarks are long running, but the algorithms are quite simple. Hence compilation is a lot faster compared to the existing test methods that feature more complex algorithms that are not looped.

**Comparing the time spent in the passes** (see table 5.5) it can be seen that MachineInstructionSchedulingPass takes more time when pattern matching is performed. LivetimeAnalysisPass is performed faster due to fewer registers used in the LIR when pattern matching is done. LinearScanAllocatorPass performance is improved for all shown benchmarks except for MulImm and LEA1. [8]

| Benchmark | Iterative Lowering | Pattern Matching | Comparison |
|---:|---:|---:|---:|
| | $\mu s$ | $\mu s$ | $1 - IL/PM$ in % |
| AddImm | 1040 | 999 | -4.08 |
| MulImm | 836 | 936 | 10.71 |
| LEA1 | 909 | 1013 | 10.26 |
| LEA2 | 906 | 1026 | 11.65 |
| LEA3 | 919 | 993 | 7.43 |
| LEA4 | 923 | 1015 | 9.08 |
| LEA5 | 961 | 1029 | 6.59 |
| LEA6 | 957 | 995 | 3.80 |
| boyerMoore | 8085 | 7991 | -1.18 |
| sqrt | 1330 | 1324 | -0.49 |
| permut | 3400 | 3722 | 8.66 |
| matAdd | 3659 | 3900 | 6.18 |
| matMult | 4749 | 4824 | 1.54 |
| matTrans | 7335 | 7642 | 4.02 |

Table 5.4: Comparison of compilation time (rounded to $\mu s$)

[8] RegisterAllocatorPass is not listed as it is a meta-pass and does not consume runtime.

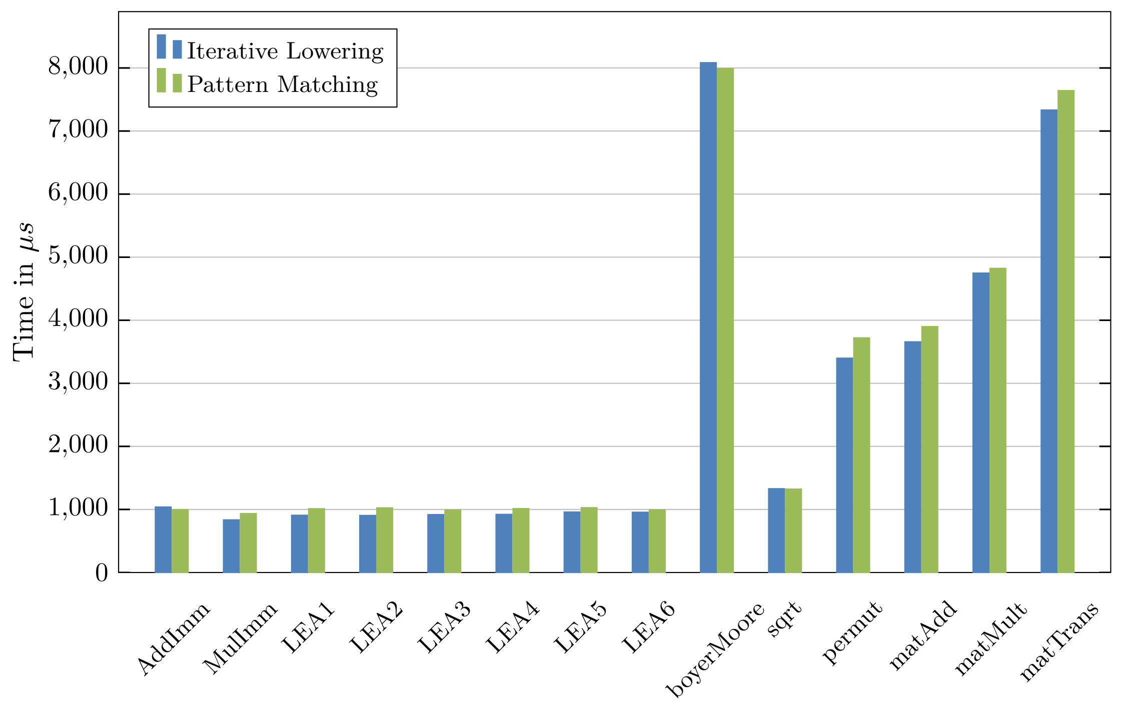


Figure 5.2: Comparison of compilation time between Iterative Lowering and Pattern Matching

| | AddImm | | MulImm | | LEA1 | | LEA5 | | boyerMoore | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pass | IL | PM | IL | PM | IL | PM | IL | PM | IL | PM |
| MachineInstructionScheduling | 222 | 300 | 166 | 244 | 187 | 292 | 201 | 299 | 2463 | 2680 |
| MachineLoop | 33 | 33 | 32 | 35 | 31 | 32 | 32 | 32 | 353 | 366 |
| LivetimeAnalysis | 75 | 58 | 61 | 54 | 72 | 58 | 75 | 58 | 310 | 279 |
| LinearScanAllocator | 472 | 370 | 364 | 377 | 400 | 404 | 429 | 405 | 3355 | 3212 |
| CodeGen | 32 | 28 | 32 | 29 | 28 | 31 | 31 | 30 | 90 | 83 |

Table 5.5: Comparison of compiler pass time consumption for selected benchmarks. All values in $\mu s$

### 5.3.3 Total Memory Allocated

Total memory allocated has not changed much (see 5.6) and is 0.46% higher on average. The AddImm benchmark uses 13% less memory, MulImm 9% less, but most other benchmarks cause the compiler to allocate slightly more memory. Compiling methods handling matrices allocated between 7.05% and 9.21% more memory. Table 5.7 shows the total memory allocated for the classes that account for most memory. Allocations for Value have increased which is directly related to creating stubs for pattern matching (NoInst is a subclass of Value). MachineOperand and MachineInstruction allocations have decreased due to fewer copy operations and complex machine instructions that replace multiple instructions. Encoding immediates instead of moving them to VirtualRegisters not only reduces MachineOperands, but also allocations of LivetimeIntervalImpl. As less values are to be organized, the lifetime analysis accounts for less allocations (and the corresponding LivetimeAnalysisPass also uses less runtime; see table 5.5).

| Benchmark | Iterative Lowering | Pattern Matching |
|---:|---:|---:|
| AddImm | 30504 | 26944 |
| MulImm | 24456 | 22400 |
| LEA1 | 27048 | 27248 |
| LEA2 | 27048 | 27608 |
| LEA3 | 27048 | 27608 |
| LEA4 | 27488 | 27248 |
| LEA5 | 28776 | 27952 |
| LEA6 | 28776 | 27952 |
| boyerMoore | 170752 | 173728 |
| sqrt | 35848 | 36600 |
| permut | 87648 | 91064 |
| matAdd | 94360 | 101512 |
| matMult | 110192 | 118552 |
| matTrans | 162328 | 178800 |

Table 5.6: Comparison of total memory allocated (in *byte*) in the optimizing compiler

| Benchmark | Strategy | Value | Machine Operand | Machine Instruction | Lifetime IntervalImpl |
|---|---|---|---|---|---|
| AddImm | IL | 5016 | 15776 | 4736 | 2392 |
| | PM | 6024 | 12208 | 4584 | 1456 |
| MulImm | IL | 3984 | 12504 | 3760 | 1560 |
| | PM | 4824 | 10936 | 2920 | 1144 |
| LEA1 | IL | 4488 | 14032 | 3896 | 2080 |
| | PM | 6168 | 12824 | 3912 | 1768 |
| LEA5 | IL | 4832 | 14848 | 4256 | 2288 |
| | PM | 6512 | 12824 | 4272 | 1768 |
| matTrans | IL | 28688 | 86968 | 32160 | 144 |
| | PM | 44144 | 86624 | 33480 | 256 |

Table 5.7: Total Memory Allocation per Class (in *byte*)

### 5.3.4 Register Allocator Performance

Comparing register allocation showed mixed results. In most cases the spill statistics are equal or differ by one spill load / store. Benchmark matTrans has far more spill loads and stores when pattern matching is active. Fewer instructions and operands should help the register allocator but in this case the opposite happens. Most likely the scheduling algorithm orders the machine instructions in an unfavorable way for register allocation. Figures 5.3 and 5.4 show the spill statistics for all benchmarks that have spill loads or stores.

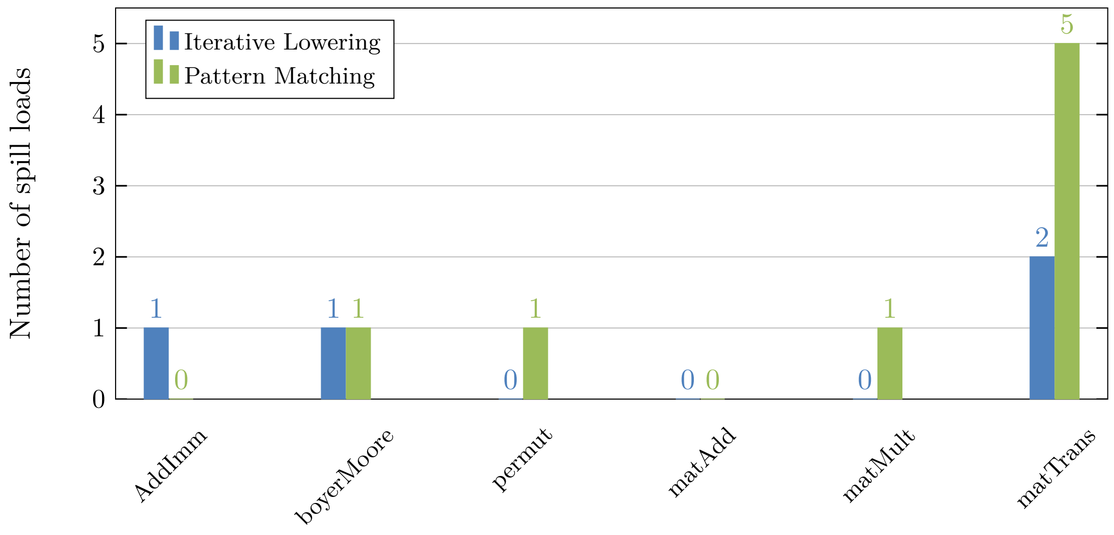


Figure 5.3: Spill Loads Comparison

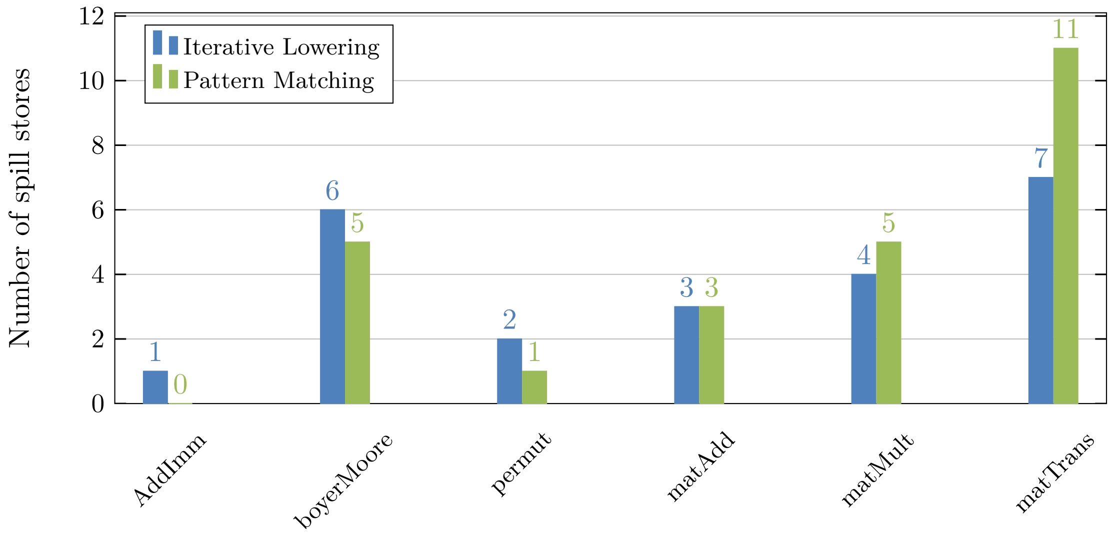


Figure 5.4: Spill Stores Comparison

### 5.3.5 Code Size

The code size of the compiled methods naturally decreases if less machine instructions are emitted (see table 5.8). Encoding immediates increases code size, but has no influence in this comparison: Traditional executables might provide constant values outside the code block (e.g. text block) and perform a load from memory, but this is not done here. The iterative lowering encodes the immediates as well, but only into a MOV instruction that moves the immediate to a register. So, encoding immediates does not change code size, but omitting MOV instructions and combining instructions (e.g. with LEA) does help to decrease it.

Only matMult and matTrans showed an increase in code size. As can be seen in section 5.3.4 the spill loads and stores have increased for these benchmarks, which means that also more spill code is generated.

| Benchmark | Iterative Lowering | Pattern Matching |
|---:|---:|---:|
| AddImm | 96 | 72 |
| MulImm | 68 | 52 |
| LEA1 | 56 | 50 |
| LEA2 | 58 | 58 |
| LEA3 | 60 | 56 |
| LEA4 | 58 | 48 |
| LEA5 | 68 | 58 |
| LEA6 | 68 | 58 |
| boyerMoore | 704 | 640 |
| sqrt | 116 | 110 |
| permut | 530 | 472 |
| matAdd | 484 | 460 |
| matMult | 576 | 580 |
| matTrans | 918 | 954 |

Table 5.8: Comparison of Code Size (in *byte*)

CHAPTER 6

# Critical Reflection

## 6.1 Limitations

The implemented tree pattern matching approach has several limitations. Hence, some processor features cannot be utilized.

**Basic Block Scope:** Sometimes nodes outside of a basic block could still be used for optimizations. E.g. a CONSTInst located in another basic block could still be encoded as immediate. Although certain exceptions could be made, this approach would be error prone. Instead instruction selection across basic block boundaries would be favorable.

**Conditional Instructions:** Some architectures (e.g. ARM) allow conditional execution of single instruction. Depending on the flags set by the previous instruction the following instruction will be executed or not. This mode reduces the jumps and labels emitted in machine code. The instruction will be executed (pipelined) and a rollback will be performed if it should not be executed, which is usually faster than a regular branch hazard. Again, this cannot be solved optimally under basic block scope as the IFInst results in 3 basic blocks (the basic block where $if$ is scheduled, the *then*- and the *else*-block).

**Packed Instructions** are instructions that are usually found in multimedia instruction set extensions (e.g. SSE for x86_64). They operate on large registers (e.g. 128bit) that are treated like a list of smaller registers (e.g. $4 * 32$ bit or $8 * 16$ bit). The instruction is then performed multiple times in parallel.

These operations are difficult to utilize. Usually loop unrolling has to be done first until multiple calculations of one loop can be combined to a packed instruction. As these operations do not share data dependencies but are independent of each other, pattern matching cannot be used.

Implementing this optimization in the backend might be difficult, as the LIR will be harder to analyze. On the other hand, an HIR optimization pass would require knowledge about the target architecture in the compiler frontend.

## 6.2 Recommended Improvements

The implementation should be improved in following aspects.

**Constraints:** The current approach misuses the cost calculation for applying constraints (see section 4.2). This is a valid solution, but it can be error prone. If an optimizing rule cannot be applied $MAX\_COST$ will be returned. But if the non-optimizing rules are missing, the optimizing rule is the only rule matching and trivially the cheapest rule to be selected. Hence, the pattern matcher will select it although it cannot be applied. The lowering code has to assert viability by again checking the same conditions that already have been checked by the cost calculation. The implementation should be extended by constraints, which would require an extension of the input grammar and the generator.

**Scheduling:** As the main task was to improve instruction selection, the scheduling approach is still pretty basic. Currently it only creates a valid schedule that is far from optimal. The chosen nested approach (scheduling the trees and scheduling the instructions within the tree; see section 3.8.5) leads to a fixed schedule that only takes care of data dependencies. It should be replaced by a fully fledged scheduling concept.

CHAPTER 7

# Summary and Future Work

This chapter provides a short summary of the work and discusses possible future work.

## 7.1 Summary

In this work the optimizing compiler of the *CACAO VM* is improved regarding instruction selection. Tree pattern matching is applied to detect patterns that can be solved efficiently by the target processor's instruction set. The implementation shows how a DAG IR can be augmented and fed into an algorithm that expects trees without duplicating parts of the intermediate representation. The optimizations done show promising evaluation results and performance is expected to improve further once other optimizations are implemented.

## 7.2 Future Work

The implemented optimizations only provide a proof-of-concept. Although some optimizations cannot be done with this approach (see section 6) providing a meaningful implementation is still an elaborate task to be done. Some promising optimizations can only be done for a more mature optimizing compiler, e.g. encoding address arithmetic using Ertl's approach (see section 2.4.2). Extending the tree grammar rules by constraints would be advisable.

Instruction scheduling clearly needs improvements and could provide significant performance gains. An optimized scheduling algorithm should not rely on the tree patterns found in the HIR. Instead it should analyze the generated machine instructions, its operands, and results and be able to move all machine instructions freely within the basic block. Further it should keep track of register pressure to produce a schedule that prevents register spills.

APPENDIX A

# Optimization Example

The optimization chapter (see section 4) only discusses the patterns but omits code that has to be written for the optimization. This section shows how an optimization is implemented in actual code. It uses a LEA optimization as example and shows cost calculation as well as lowering code.

## A.1 Rules and Cost Calculation

Listing A.1 shows a LEA optimization pattern with all operands being used. There are several ways the HIR can be structured when calculating the same value due to commutative operations, only one pattern is discussed here.

The string token `BaseIndexMultiplierDisplacement` is the identifier of the rule that is used in the lowering afterwards. Even if different rules can be lowered in the same way the token must not be reused. All string tokens are collected in an enum and compilation will fail if a name is specified more than once.

The operands of an Instruction are only available as Value supertype. Hence checking properties of a subtype requires calling the *virtual casting functions*.

```
...
stm: ADDInstID(ADDInstID(stm, MULInstID(stm, CONSTInstID)),
  CONSTInstID) "BaseIndexMultiplierDisplacement"
  calcBaseIndexMultiplierDisplacementCost(a)
...
%%

namespace {

bool isMultiplier(CONSTInst* c){
    s8 val = c->get_value();
    if ((val == 2) || (val == 4) || (val == 8)) return true;
    return false;
}

bool isConstEncodable(CONSTInst* c){
    return fits_into<s4>(c->get_value());
}

bool isDiscreteValue(Instruction* a){
    return ( a->get_type() == Type::ByteTypeID  ||
            a->get_type() == Type::IntTypeID    ||
            a->get_type() == Type::LongTypeID);
}

int calcBaseIndexMultiplierDisplacementCost(Instruction* a){
    if (isDiscreteValue(a) &&
        isConstEncodable(
            a->get_operand(1)->to_Instruction()->to_CONSTInst())
             &&
        isMultiplier(
            a->get_operand(0)->to_Instruction()->get_operand(1)
                ->to_Instruction()->get_operand(1)
                ->to_Instruction()->to_CONSTInst()))
        return 1;
    return MAX_COST;
}

}
```

Listing A.1: LEA Pattern and Cost Calculation

## A.2 Lowering

The lowerComplex method in the X86_64LoweringVisitor is basically a big switch/case statement and for every pattern a case block is available that performs the lowering. Again dynamic casts are required to access properties, e.g. constant values of CONSTInst.

Up to line 16 in listing A.2 the HIR instructions are *reconstructed* and MachineOperands are gathered for the *tree foreign* values, e.g. the `base` operand in line 10 from an arbitrary HIR instruction that is referenced. In line 18 a VirtualRegister is created that will hold the result. Line 19 shows the creation of a ModRMOperand that takes all the input operands that will later be encoded as ModRM and SIB byte (see [Int]). In line 21 the LEAInst is created and the following line pushes it into the code memory.

At last, the result (i.e. the VirtualRegister created in line 18) is linked to *the* HIR instruction, i.e. the root instruction of the tree pattern. This way other HIR instructions that use the defining HIR instruction as operand can find the MachineOperand that is used as LIR input. See line 10 on how a user can access this information.

```
 1 void X86_64LoweringVisitor::lowerComplex(Instruction* I, int
     ruleId){
 2     switch(ruleId){
 3         // other rules ...
 4         case BaseIndexMultiplierDisplacement:
 5         {
 6             assert(I);
 7             Type::TypeID type = I->get_type();
 8
 9             Instruction* bim_root =
                 I->get_operand(0)->to_Instruction();
10             MachineOperand* base =
                 get_op(bim_root->get_operand(0)->to_Instruction());
11
12             Instruction* nested_mul =
                 bim_root->get_operand(1)->to_Instruction();
13             MachineOperand* index =
                 get_op(nested_mul->get_operand(0)->to_Instruction());
14             CONSTInst* multiplier = nested_mul->get_operand(1)
15                 ->to_Instruction()->to_CONSTInst();
16
17             CONSTInst* displacement =
                 I->get_operand(1)->to_Instruction()->to_CONSTInst();
18
19             VirtualRegister *dst = new VirtualRegister(type);
20             ModRMOperand
                 modrm(ModRMOperand::get_scale(multiplier->get_Int()),
                 IndexOp(index), BaseOp(base),
                 displacement->get_value());
21
22             MachineInstruction* lea = new LEAInst(DstOp(dst),
                 get_OperandSize_from_Type(type), SrcModRM(modrm));
23             get_current()->push_back(lea);
24             set_op(I,lea->get_result().op);
25         }
26         break;
27         default:
28             ABORT_MSG("Rule␣not␣supported", "Rule␣" << ruleId
                 << "␣is␣not␣supported␣by␣method␣lowerComplex!");
```

```

    }
}
```

Listing A.2: Lowering Code for Optimizing Rules

## A.3 LIR Code

Listing A.3 shows the LEAInst class definition and the `emit` method implementation that handles the encoding of instruction opcode, registers and immediate values. The content of the LEAInst class and its `emit` method is mainly copied from MovModRMInst, as MOV uses the same address calculation method as LEA. Only newly created code is shown, so ModRMOperandDesc and called functions are not shown in the listings.

```
class LEAInst : public GPInstruction {
private:
    enum OpIndex {
        Base = 0,
        Index = 1,
        Value = 2
    };
    ModRMOperandDesc modrm;
public:
    LEAInst( const DstOp &dst, OperandSize op_size, const
      SrcModRM src )
        : GPInstruction("X86_64LEAInst", dst.op, op_size, 2),
            modrm( ModRMOperandDesc(
              src.op.scale,operands[Index], operands[Base],
              src.op.disp )){
        operands[Base].op = src.op.base;
        operands[Index].op = src.op.index;
    }
    virtual void emit(CodeMemory* CM) const;
};

void LEAInst::emit(CodeMemory* CM) const {

    X86_64Register *reg;
    u1 opcode;

    MachineOperand *op = get_result().op;
    reg = (op?cast_to<X86_64Register>(op) : 0);

    opcode = 0x8D;

    assert(reg);
    CodeSegmentBuilder code;
    // set rex
    u1 rex = get_rex(reg,modrm,get_op_size() ==
      GPInstruction::OS_64);
    if (rex != 0x40)
        code += rex;
    // set opcode
    code += opcode;
    // set modrm byte
    // mod
    u1 modrm_mod;
    if (modrm.disp == 0) {
        // no disp
        modrm_mod = 0;
    }
    else if (fits_into<s1>(modrm.disp)) {
        // disp8
        modrm_mod = 1;
    } else {
        // disp32
        modrm_mod = 2;
    }
    X86_64Register *index_reg = (modrm.index.op != &NoOperand
      ? cast_to<X86_64Register>(modrm.index.op) : 0);
    X86_64Register *base_reg  = (modrm.base.op  != &NoOperand
      ? cast_to<X86_64Register>(modrm.base.op ) : 0);
    // r/m
    u1 modrm_rm;
    bool need_sib;
    // for the time being always use SIB byte
    if (!index_reg && base_reg && base_reg->get_index() !=
      0x4) { // 0b100
        modrm_rm = base_reg->get_index();
        need_sib = false;
    } else {
        modrm_rm = 4; // 0b100
        need_sib = true;
    }
    code += get_modrm_u1(modrm_mod,reg->get_index(),modrm_rm);

    if (need_sib) {
        // set sib
        u1 sib=0;
        sib |= modrm.scale << 6;
        if(index_reg) {
            sib |= (0x7 & index_reg->get_index()) << 3;
        }
        else {
            sib |= 0x4 << 3; // 0b100 << 3
        }
        assert(base_reg);
        sib |= (0x7 & base_reg->get_index()) << 0 ;
        code += sib;
    }
    if (modrm_mod == 1) {
        code += s1(modrm.disp);
    }
    if (modrm_mod == 2) {
        code += (0xff && (modrm.disp >>  0));
        code += (0xff && (modrm.disp >>  8));
        code += (0xff && (modrm.disp >> 16));
        code += (0xff && (modrm.disp >> 24));
    }
    add_CodeSegmentBuilder(CM,code);
}
```

Listing A.3: LEAInst Code

# Bibliography

[AGH05] Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, October 1989. URL: `http://doi.acm.org/10.1145/69558.75700`.

[AP03] Andrew W Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, 1988. URL: `http://doi.acm.org/10.1145/73560.73561`.

[BCKT89] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *SIGPLAN Not.*, 24(7):275–284, June 1989. URL: `http://doi.acm.org/10.1145/74818.74843`.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: `http://doi.acm.org/10.1145/115372.115320`.

[CH90] Fred C. Chow and John L. Hennessy. The Priority-based Coloring Approach to Register Allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October 1990. URL: `http://doi.acm.org/10.1145/88616.88621`.

[Cha82] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982. URL: `http://doi.acm.org/10.1145/872726.806984`.

[Cli95] Cliff Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

[CP95] Cliff Click and Michael Paleczny. A Simple Graph-based Intermediate Representation. *SIGPLAN Not.*, 30(3):35–49, March 1995. URL: `http://doi.acm.org/10.1145/202530.202534`.

[DaC] DaCapo Project. DaCapo Benchmarks. Last visited 2015-03-19. URL: `http://www.dacapobench.org`.

[DSW+13] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[DWS+13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, 2013. URL: `http://doi.acm.org/10.1145/2542142.2542143`.

[EBS+08] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized Instruction Selection Using SSA-graphs. *SIGPLAN Not.*, 43(7):31–40, June 2008. URL: `http://doi.acm.org/10.1145/1379023.1375663`.

[Eis13] Josef Eisl. Optimization Framework for the CACAO VM. Master's thesis, TU Vienna, 2013.

[EKS03] Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection Based on SSA-Graphs. In Andreas Krall, editor, *Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

[Ert99] M. Anton Ertl. Optimal Code Selection in DAGs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 242–249, 1999. URL: `http://doi.acm.org/10.1145/292540.292562`.

[ESL89] H. Emmelmann, F.-W. Schröer, and Rudolf Landwehr. BEG: A Generator for Efficient Back Ends. *SIGPLAN Not.*, 24(7):227–237, June 1989. URL: `http://doi.acm.org/10.1145/74818.74838`.

[ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference. URL: `http://www.complang.tuwien.ac.at/papers/ertl+06dotnet.ps.gz`.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[FHP92a] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code-generator Generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, September 1992. URL: `http://doi.acm.org/10.1145/151640.151642`.

[FHP92b] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not.*, 27(4):68–76, April 1992. URL: `http://doi.acm.org/10.1145/131080.131089`.

[GH88] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 442–452, 1988. URL: `http://doi.acm.org/10.1145/55364.55407`.

[GM86] Philip B. Gibbons and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. *SIGPLAN Not.*, 21(7):11–16, July 1986. URL: `http://doi.acm.org/10.1145/13310.13312`.

[GO] Shay Gal-On. CoreMark. Last visited 2015-03-19. URL: `http://www.eembc.org/coremark/`.

[Int] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Last visited 2015-03-19. URL: `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[KG97] Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[KG08] David Ryan Koes and Seth Copen Goldstein. Near-optimal Instruction Selection on Dags. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 45–54, 2008. URL: `http://doi.acm.org/10.1145/1356058.1356065`.

[Kra98] 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. URL: `http://dl.acm.org/citation.cfm?id=522344.825703`.

[KWM$^+$08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™

Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. URL: `http://doi.acm.org/10.1145/1369396.1370017`.

[LA04a] Chris Lattner and Vikram 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. URL: `http://dl.acm.org/citation.cfm?id=977395.977673`.

[LA04b] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, September 2004.

[LLV] LLVM Compiler Infrastructure Project. LLVM Documentation. Last visited 2015-03-19. URL: `http://llvm.org/docs/`.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[Nov03] Diego Novillo. Tree SSA A New Optimization Infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193. Citeseer, 2003.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999. URL: `http://doi.acm.org/10.1145/330249.330250`.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff 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`.

[Rau78] B. Ramakrishna Rau. Levels of Representation of Programs and the Architecture of Universal Host Machines. *SIGMICRO Newsl.*, 9(4):67–79, November 1978. URL: `http://dl.acm.org/citation.cfm?id=1014198.804311`.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, 1988. URL: `http://doi.acm.org/10.1145/73560.73562`.

[SDMW12] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas 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, 2012. URL: `http://doi.acm.org/10.1145/2414740.2414750`.

[Sta] Standard Performance Evaluation Corporation. SPECjvm. Last visited 2015-03-19. URL: `https://www.spec.org/jvm2008/`.

[Tro] John Tromp. The Fhourstones Benchmark. Last visited 2015-03-19. URL: `http://tromp.github.io/c4/fhour.html`.

[War90] H. S. Warren, Jr. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM J. Res. Dev.*, 34(1):85–92, January 1990. URL: `http://dx.doi.org/10.1147/rd.341.0085`, `doi:10.1147/rd.341.0085`.

[Wei84] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984. URL: `http://doi.acm.org/10.1145/358274.358283`.

[WF10] Christian Wimmer and Michael 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, 2010. URL: `http://doi.acm.org/10.1145/1772954.1772979`.

[WM05] Christian Wimmer and Hanspeter 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, 2005. URL: `http://doi.acm.org/10.1145/1064979.1064998`.