TU WIEN Informatics

Fast machine-code generation for stack-based languages

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Siegfried Oleg Pammer
Registration Number 01633095

to the Faculty of Informatics
at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Vienna, 27th April, 2023

Siegfried Oleg Pammer | M. Anton Ertl

Erklärung zur Verfassung der Arbeit

Siegfried Oleg Pammer

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, 27. April 2023

Siegfried Oleg Pammer

Danksagung

Mein hauptsächlicher Dank gilt meinem Betreuer M. Anton Ertl, für die Unterstützung und Begleitung während dieser Arbeit.

Dank gebührt außerdem John Witulski und Daniel Grunwald für die hilfreichen Kommentare zu meiner Arbeit, und der ic#code Gruppe für die Erfahrungen, die ich in den letzten 15 Jahren machen konnte, und dass sie mein Interesse an Übersetzerbau und verwandten Themen geweckt haben.

Acknowledgements

Mainly I want to thank my advisor M. Anton Ertl for his support during the writing of this thesis.

I want to also thank John Witulski and Daniel Grunwald for their helpful comments, and the ic#code group for the experience I could gain during the last 15 years, and that they sparked my interest in compiler construction and related topics.

Kurzfassung

In dieser Arbeit werden Implementierungstechniken für schnelle Befehlsauswahl im Kontext von stapelbasierten Sprachen untersucht. Ausgehend von Tree-Parsing und der Implementierung von `iburg` wird untersucht, wie die Flaschenhälse des Tree-Parsing-Ansatzes von `iburg` gelöst werden können: Die Notwendigkeit, Ausdrucksbäume zu konstruieren und die Notwendigkeit eines Algorithmus mit zwei Durchläufen. Als eine mögliche Lösung wird ein Generator implementiert, der aus den Tree-Parsing-Regeln und den zugehörigen Aktionen einen endlichen Automaten erzeugt, der nur einen einzigen Durchlauf und keine Baum-Konstruktion benötigt. Die Leistungsfähigkeit eines hangeschriebenen Codegenerators, des Tree-Parsings von `iburg` und eines automatenbasierten Ansatzes wird gemessen und verglichen. Der generierte Automat liefert ähnlich gute Ergebnisse, wie ein handgeschriebener Codegenerator.

Abstract

This work investigates implementation techniques for fast instruction selection in the context of stack-based languages. Using tree-parsing and the `iburg` implementation thereof as a starting point, this work is looking into solving the bottle-necks of `iburg`'s tree-parsing approach: The need to construct expression trees and the need for a two-pass algorithm. This work implements a generator that produces a finite-state machine from tree-parsing rules and associated actions, which only requires a single pass and no tree-construction. The performance of a manually written code generator, `iburg`'s tree parsing and an automaton-based approach is measured and compared. The generated automaton produces results of similar quality as a hand-written code generator.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation and aim 2
1.2 The CACAO JVM 3

2 Related Work **5**
2.1 Code generation 5
2.2 Tree-parsing and the `burg` family 5

3 Background **7**
3.1 Stack-based languages and Virtual Machines 7
3.2 The Java VM and its Byte-code 9
3.3 Cacao JVM 14
3.4 Tree-parsing, `(i)burg` and `bfe` 16

4 Approaches under Examination **25**
4.1 Tree-parsing Automata using Dynamic Programming 27
4.2 Naïve Expansion 30
4.3 Automata 31
4.4 Pushdown Automata 37

5 Results **39**
5.1 Compiler execution time 39
5.2 Generated code size 40
5.3 Total execution time 41
5.4 Instruction counts and applied rules 42
5.5 Reset hit counts 45

6 Further Work **51**

7 Conclusion **53**

Bibliography **55**

List of Figures **58**

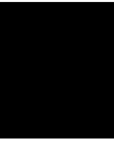
List of Tables **59**

Appendix **63**

A.1 `awk` script used to generate the reducer 63

A.2 Source code of custom Java benchmarks 65

CHAPTER 1



Introduction

In software development compilers play an important role by making it possible to use high-level languages, which are easily understood by humans, instead of having to encode algorithms directly in binary hardware instructions. Another advantage is that programs can be implemented once and then be compiled for multiple target platforms.

Compilers are programs that take an input program in a given input language and compile it to a given output language, which can then be executed by a computer. In addition to compilers, there are also *interpreters*, which can directly execute programs in a given language. Compilation usually happens *ahead-of-time* (AOT) and the generated machine-code can be directly executed on a compatible target platform, while interpreters need to be distributed alongside the program to the target computer for the program to be executed.

Interpreters have the advantage, that they can be implemented with lower effort than a compiler, but the execution of the program is slower. Compilers are harder to implement, but can perform many optimizations and utilize machine-code directly, which leads to faster execution of the program.

Compilers can be divided into two categories:

- Ahead-Of-Time Compilation (AOT): Programs in a high-level language are translated to the target machine-code as a whole and then executed. This allows for a variety of optimizations to be applied, such as dead-code elimination, inlining and the use of target-specific special-purpose instructions. The advantage is that the machine-code is very fast and also *time-to-execution* is very short. A downside is that the program may require recompilation, if some parts of the target computer (software or hardware) change.
- Just-In-Time Compilation (JIT): Programs in a high-level language are first translated to an intermediate representation (AOT) and then translated to the target

> machine-code by the JIT-compiler at run-time before execution. This is a hybrid approach combining the advantages of both interpretation and AOT-compilation. It allows for optimizations to be applied prior to execution, but also avoids having to recompile the program for other architectures, as long as there is a JIT-compiler for the new target architecture. However, because the compilation is now happening right before execution, the *time-to-execution* increases and the challenge is to produce fast and small code, while also not increasing *time-to-execution* too much. The JIT-compiler may also use its "run-time knowledge" to perform so-called profile-guided optimizations (PGOs), which optimize parts of a program that are most used, based on statistics collected at run-time. This is typically done in two stages: At the first stage the code is compiled as quickly as possible without special optimizations. If the code turns out to be executed multiple times, the code is recompiled (stage 2) and optimizations are applied.

In order to produce efficient code (efficiently), compilers and interpreters have to solve a number of problems, one of which is instruction selection. Its main purpose is to transform instructions from a high-level intermediate representation (IR) into a low-level representation or target machine-code. The goal is to select the best code sequence "in terms of speed, size or some other metric, so we have an optimization problem [...]"[4, pg. viii] After (optimal) machine-code instructions have been selected, other steps such as instruction scheduling and register allocation may be performed.

The topic has received broad attention from the scientific community over the last 60 years. In 2013 G. Blindell [4] compiled an extensive summary of known approaches and literature. It lists four general approaches to instruction selection: Macro expansion, tree covering, DAG covering and other graph-based approaches. This work will focus on tree covering as implemented by the `burg` instruction selector generator family in the context of stack-based languages.

1.1 Motivation and aim

Instruction selection using tree-parsing is solvable in linear time, however, as code generators should produce code as fast as possible, looking for a faster solution is important. The instruction selectors generated by `burg` (automaton-based) and `iburg` (dynamic-programming-based) consist of two parts: the *labeller* and *reducer* and both need to traverse the tree, which is expensive. The automaton-based approach is faster than the approach based on dynamic-programming, but the generated automata are quite big (depending on the input tree grammar).

This work seeks to answer the following question: Can a faster instruction selector be generated from tree grammars for stack-based intermediate languages? How much code quality must be sacrificed to gain this benefit?

The aim of this work is to implement and compare handwritten (naïve) expansion (as base-line) and different implementation strategies for tree covering instruction selectors:

iburg's (dynamic-programming-based) tree pattern-matching, (finite-state) automata and pushdown automata. The latter two will be implemented as part of this work. The criteria for comparison are compilation and execution time and generated machine-code size of chosen benchmarks.

The main contributions of this work are:

- We present a way to generate one-pass finite-state automata for stack-based (intermediate) languages from tree-parsing grammars.
- We present measurements and a comparison of a hand-written instruction selector, burg's tree pattern-matching, our finite-state automata and pushdown automata.

1.2 The CACAO JVM

In order to allow for realistic evaluation and comparisons, the approaches will be implemented as part of the CACAO JVM,[1] an open-source Java virtual machine implementation developed at TU Wien. Development started in 1996 and the primary goal was to implement the fastest Just-In-Time compiler at the time for the Alpha architecture.

In the following years support for further target architectures was added:

- ARM
- MIPS (32 and 64 bit)
- PowerPC (32 and 64 bit)
- System/390
- SPARC 64
- x86
- AMD64

In this work the AMD64 architecture is used for all measurements.

[1] http://www.cacaojvm.org/

CHAPTER 2

Related Work

2.1 Code generation

Code generation for register-based machines was proven to be NP-complete by Bruno and Sethi [6] in 1976. Optimal translation from (expression) trees into code for register machines is discussed by Aho and Johnson [1]. They discuss optimality of expression-tree-to-register translation and present an algorithm based on dynamic programming, which is able to produce optimal code.

In 1989 [2] Aho et al. introduce `Twig`, a tree-pattern-matching code-generator generator based on dynamic programming, which generates code in two passes. The first pass finds minimum-cost patterns and the second pass executes semantic actions associated with the states, i.e., emits an optimal instruction sequence.

Emmelmann et al., [9] present `BEG` which also relies on tree-pattern-matching.

2.2 Tree-parsing and the `burg` family

Balachandran et al. [3] propose a system for optimal instruction selection using tree-pattern-matching for expression trees. A subset of it is used by Fraser et al. [13] to create `burg`, which is similar `Twig` but less expressive and faster. A predecessor of `burg` was published by David R. Chase [8], which only supports one non-terminal symbol. After the introduction of `burg`, there have been many different implementations named after `burg`, although they often only bear a similar name:

- `iburg` is introduced by Fraser et al.[12], which is a simplified version of `burg` which uses dynamic programming instead of an automaton.
- `lcc` is a retargetable C compiler[11], which uses `lburg` a variant of `iburg` that allows for dynamic costs.

- `mburg` by Gough [15] is an implementation of `iburg` using Modula-2, which performs consistency checks on the tree-grammar and allows adding `YACC`-style declarative actions directly in the grammar specification.
- Frazer and Proebsting [14] create `gburg` where they apply the `burg` "system" to stack-based interpreters/JIT compilers. It takes advantage of a stack-based system and does not operate on trees directly, but a linearized postfix notation, i.e., "stack code" of them. The one-pass code generator uses a greedy algorithm, which is not guaranteed to be optimal. The approach presented in this work is similar to `gburg`, but does not require a separate "construction grammar". A comparison of this work with `gburg` is presented in section 4.3.5.
- `wburg` is another improvement to `burg` presented by Proebsting, et al. [21], which allows to generate a generator that just requires a single pass.

Ertl et al., [10] present a combination of tree-parsing automata and dynamic-programming instruction selectors, which generates a tree-parsing automaton on demand at JIT compilation time.

CHAPTER 3

Background

3.1 Stack-based languages and Virtual Machines

Stack-based languages are languages with one (or more) programmer-visible stacks used for passing operands/arguments to operations and procedures/methods and for returning results.

A *stack* is a data structure that allows storage of multiple items, similar to a list, however, it is only possible to insert or remove items at the top (often referred to as *TOS* = "top of stack"), similar to a stack of dishes in a kitchen. The operation of inserting an item is called *push* and removing an item from the stack is called *pop*. See figure 3.1 for an illustration.

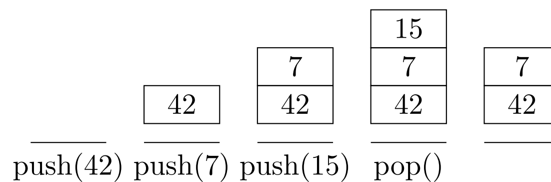


Figure 3.1: Diagram of a stack, showing push and pop operations. Each step shows the stack before the operation, above the next operation.

The concept was first introduced to the field of computer science by Alan M. Turing in 1945 [23, pg. 11/12], where he used it to model subroutine calls and returns. He used the terms "bury" and "unbury" for the *push* and *pop* operations, respectively.

In the 1960s Reverse Polish[1] notation (RPN) was first used to describe stack-based expression evaluation, as the order of notation matches the order of evaluation, because

[1] Named after Jan Łukasiewicz, a Polish logician and philosopher.

each operator directly works with the results of the preceding operations. For example, $x + 1$ is written in post-fix notation as `x 1 +`. An expression such as `(x + 1) * 5`, requiring parentheses, is written as `x 1 + 5 *`. RPN was used in handheld calculators[2] and by stack-based programming languages, most notably by Forth and Postscript.

The term "stack-effect" describes the effect of operations on the stack. In this work, a notation similar to the Forth stack-effect notation[3] is used: `count of items removed from the stack - count of items added to the stack`. For example, the stack-effect of integer addition would be written as `2 - 1`, because addition consumes two integers from the evaluation stack and adds/pushes the sum of them back onto the stack.

While not seeing wide-spread adoption in programming languages for humans, stack-based programming paradigms are extensively used in *process virtual machines*, also called "abstract machines", "programming language VMs" or Managed Runtime Environments (MREs).

3.1.1 Process virtual machines or Programming language virtual machine

"A process VM is a virtual platform created for an individual process and destroyed once the process terminates. Virtually all operating systems provide a process VM for each one of the applications running, but the more interesting process VMs are those which support binaries compiled on a different instruction set."[19, 10.2, p. 367]

Many such VMs use a stack-based execution model and intermediate language (examples include the Java VM, CLR and the Python VM), which is then interpreted directly or JIT-compiled for the target - often register-based - hardware architecture. As this work focuses on Java, a detailed description of the Java VM will be given in the next section.

However, there are also VMs that use register-based intermediate languages, such as the Dalvik virtual machine for Android, which compiles JVM byte-code to register-based byte-code called "Dalvik VM byte-code", and the Lua 5.0 virtual machine. Ierusalimschy et al. [17] and D. Gregg et al. [16] argue that register-based VMs are superior to stack-based machines, because they avoid the overhead that comes with the necessary stack management. Another advantage is that register-based machines need fewer instructions to achieve the same result, which can be seen in the following example, which compares Java source code, JVM byte-code and Dalvik VM byte-code:

Java source code:	`int b = a + 5;`	*Java VM* byte-code (stack-based):

[2] An advantage of reverse polish notation (post-fix notation) over traditional infix notation is that parentheses are not required, which leads to a reduction of keystrokes.

[3] `https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Stack_002dEffect-Comments-Tutorial.html` retrieved December 2022.

ILOAD_0 ICONST_5 IADD ISTORE_1	*Dalvik VM* byte-code (register-based): `add-int/lit16 v1, v0,`	`#+5`

Explanation of the instructions:

- `ILOAD_0` loads the value stored in the first 32-bit integer variable slot onto the evaluation stack. (Stack-effect: `0 - 1`)
- `ICONST_5` loads the integer constant `5` onto the evaluation stack. (Stack-effect: `0 - 1`)
- `IADD`: pops two 32-bit values off the evaluation stack and pushes their sum onto the evaluation stack. (Stack-effect: `2 - 1`)
- `ISTORE_1`: pops a 32-bit value off the evaluation stack and stores it in the second 32-bit integer variable slot. (Stack-effect: `1 - 0`)
- `add-int/lit16 reg_dest, reg_src, imm16`: Loads a value from `reg_src`, adds the 16-bit immediate value to it and stores the result in `reg_dest`.

As can be seen, the JVM byte-code consists mostly of simple single-byte instructions, without any operands - the operands are stored on the implicit evaluation stack. A stack-based VM makes fewer assumptions about the hardware architecture. The stack can be easily implemented in memory. In contrast, register-based machines "must decode their operands from the instruction. Such decoding adds overhead to the interpreter." [17] Also, the Dalvik VM supports 256 registers (up to 65536 registers in some instructions), but common hardware architectures use up to 16 registers (for example, AMD64 or ARM), so the issue of register allocation remains even for register-based byte-code.

3.2 The Java VM and its Byte-code

"To implement the Java Virtual Machine correctly, you need only be able to read the class file format and correctly perform the operations specified therein. Implementation details that are not part of the Java Virtual Machine's specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java Virtual Machine instructions (for example, translating them into machine code) are left to the discretion of the implementor." [4]

[4] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html, retrieved December 2022

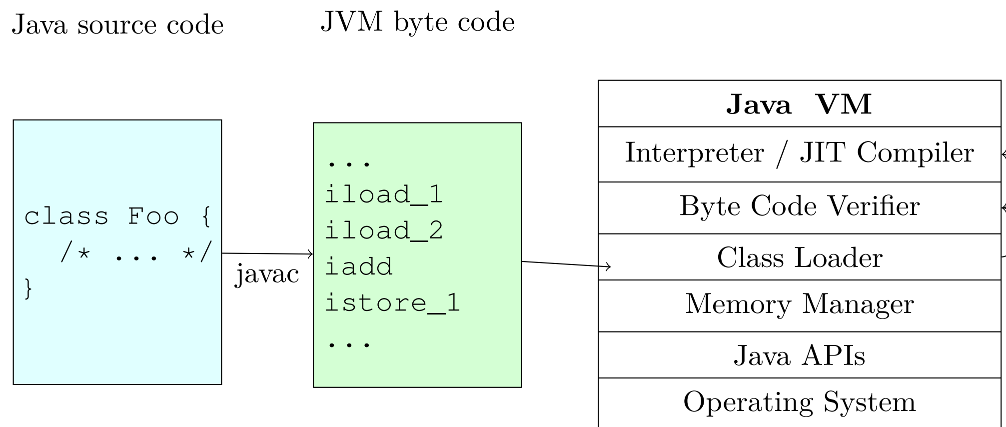


Figure 3.2: A simplified overview of the JVM architecture. It also shows that Java is compiled first into JVM byte-code using the Java compiler `javac`.

The Java Virtual Machine in general consists of the following components (see also figure 3.2):

- A *class loader*: Its responsibility is to read the contents of `class` file and then set up an in-memory representation for use by the other parts of the virtual machine. These files are, for example, produced by the Java compiler `javac` from Java source code.
- A memory manager (garbage collector), which handles allocation and deallocation of memory consumed by the program.
- A byte-code interpreter or JIT-compiler, that translates the machine-independent instructions into native machine-code at run-time.
- A byte-code verifier (optional), which ensures that the loaded byte-code adheres to the rules specified in the JVM byte-code specification.
- Because many base functions are implemented directly in JVM byte-code, the Java API is vital for the JVMs functioning and the execution of programs written in Java.

3.2.1 The byte-code interpreter/compiler

The Java byte-code may be executed using one or more of the following techniques:

- Interpreter,
- Just-In-Time compiler,

- Ahead-Of-Time compiler,
- Mixed compiler, combining one or more of the above.

The byte-code interpreter/compiler's job is to translate the byte-code to machine-specific native instructions. The Java Virtual Machine Specification does not specify how the instructions are translated, only their semantics are defined. The JVM defines the following signed integral data types: byte (8-bit), short (16-bit), int (32-bit) and long (64-bit); and the following floating-point types: float (32-bit) and double (64-bit). Additionally it supports the boolean data type, which is encoded as byte, where `0` means false and `1` true, and a char data type, an unsigned 16-bit integer representing Unicode code points. However, because the JVM is not expected to perform any type-checking and the evaluation stack is generally untyped, the compiler producing the JVM byte-code is expected to make sure, that appropriate byte-code instructions are used.

3.2.2 The JVM byte-code instructions

The JVM supports around 200 instructions, which can be grouped into the following categories: control-flow, loading and storing values, operations (arithmetic, bitwise, logical and method calls), type checking and conversion, stack manipulation, prefixes and internal instructions.

Many of the instructions are typed, which means there are separate instructions for the different integral, floating point types and the boolean type. The mnemonics for these instructions have prefixes, which describe the type they are operating on: `a` for addresses (or arrays in some cases), `b` for byte or boolean values, `c` for char(acter)s, `d` for doubles, `f` for floats, `i` for integers, `l` for longs and `s` for shorts.

The following list provides a short description of all instructions:

- 20 instructions for loading constant values onto the evaluation stack (stack-effect: 0 – 1).
- 33 `load` instructions for loading values from local variables and arrays. These instructions push a value on the evaluation stack (stack-effect: 0 – 1). The array-specific variants consume an array reference and an index from the evaluation stack (stack-effect: 2 – 1).
- 33 `store` instructions for storing values to local variables and arrays. The array-specific variants consume an array reference, an index and a value from the evaluation stack (stack-effect: 3 – 0), while the other variants only consume one single value (stack-effect: 1 – 0).
- 4 instructions for working with object constructors and arrays: `new` for creating a new object of a given type and invoke its constructor (stack-effect: 0 – 1),

newarray for creating arrays of primitive types (stack-effect: 1 – 1), anewarray for creating arrays of reference types (stack-effect: 1 – 1) and multianewarray for creating multidimensional arrays (stack-effect: 1 – 1). For all array creation instructions the length of the array is consumed from the evaluation stack.

- arraylength for retrieving the length of an array reference (stack-effect: 1 – 1).
- getfield for loading a value from an instance field onto the evaluation stack (stack-effect: 1 – 1).
- getstatic for loading a value from a static field onto the evaluation stack (stack-effect: 0 – 1).
- putfield for storing a value in an instance field (stack-effect: 2 – 0).
- putstatic for storing a value in a static field (stack-effect: 1 – 0).
- Unary arithmetic negation for each data-type: dneg, fneg, ineg and lneg. The instruction consumes one value from the evaluation stack and pushes its negated value back onto the evaluation stack.
- Integer increment iinc: This instruction does not work with the evaluation stack but instead increments the value of a local variable slot.
- Binary arithmetic operations (addition, subtraction, multiplication, division and remainder) for each data-type: dadd, dsub, dmul, ddiv, drem, fadd, fsub, fmul, fdiv, frem, iadd, isub, imul, idiv, irem, ladd, lsub, lmul, ldiv and lrem. All of them consume two items from the evaluation stack, perform the operation and push the result back onto the evaluation stack.
- Binary bit-wise operations (and, or, exclusive or, arithmetic left shift, arithmetic right shift and logical right shift) for integral data-types: iand, ior, ixor, ishl, ishr, iushr land, lor, lxor, lshl, lshr and lushr. All of them consume two items from the evaluation stack, perform the operation and push the result back onto the evaluation stack.
- 5 instructions used to invoke methods: invokestatic, invokevirtual, invokeinterface, invokespecial and invokedynamic. All of them consume the number of arguments and an object reference (for instance methods) from the evaluation stack and push the result back onto the evaluation stack (for non-void methods).
- Stack manipulation instructions:
  - dup duplicate the top stack item (stack-effect: a – a a),
  - dup2 duplicate one or two top stack items (stack-effect: a b – a b a b),

  - `dup_x1` duplicate the top stack item and insert it two items down (stack-effect: a b – b a b),
  - `dup_x2` duplicate the top stack item and insert it two or three items down (stack-effect: a b c – c a b c),
  - `dup2_x1` duplicate one or two top stack item and insert them two or three items down (stack-effect: a b c d – c d a b c d),
  - `dup2_x2` duplicate one or two top stack item and insert them two, three or four items down (stack-effect: a b c d – c d a b c d),
  - `pop` remove the top stack item (stack-effect: a –),
  - `pop2` remove one or two stack items from the top (stack-effect: a b –),
  - `swap` swap the two top stack items (stack-effect: a b – b a).

- Conversion instructions for all possible primitive type conversions: `d2f`, `d2i`, `d2l`, `f2d`, `f2i`, `f2l`, `i2b`, `i2c`, `i2d`, `i2f`, `i2l`, `i2s`, `l2d`, `l2f` and `l2i`. Note that some of these are lossy conversions, which means that information may get lost. This is especially the case with all conversions from larger to smaller types and from floating point types to integral types.

- `checkcast` and `instanceof` for working with reference types. `checkcast` tries to cast the value on top of the stack to a given reference type and throws an exception, if the cast is not possible. Null values do not cause an exception. `instanceof` also consumes the value from the evaluation stack and tries to cast it and pushes either 0 (false) or 1 (true) on to the evaluation stack, depending on whether the cast would succeed.

- Unconditional branching instructions: `goto` and `goto_w`, which unconditionally move the instruction pointer to the given offset relative to the current instruction.

- `jsr`, `jsr_w` and `ret`: Branching instructions that save/load a return address.

- Return instructions: `return`, `areturn`, `dreturn`, `freturn`, `ireturn` and `lreturn`, which end the execution of a method and may return a value to the caller. Only the current value on top of the execution stack is returned, all other values are discarded and the operand stack of the caller is restored with the returned value added on top.

- Comparison instructions: `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl` and `lcmp`: These instructions consume and compare the top two items on the evaluation stack and push `0`, if both values are equal, push `-1` if the first value is lower than the second or `1`, if the first value is greater than the second. The instructions dealing with floating point types have a special-case dealing with `NaN` values: `dcmpg`/`fcmpg` pushes `1` and `dcmpl`/`fcmpl` pushes `-1` if at least one input value is `NaN`.

- The `tableswitch` instruction implements a jump-table using a contiguous range of indices.

- The lookupswitch instruction implements a jump-table based on keys.
- The ifnull/ifnonnull instructions implement a conditional branch, if the value on the stack is null (or not null).
- The ifeq, ifne, iflt, ifge, ifgt and ifle instructions consume the value on top of the evaluation stack, perform a comparison with zero and branch to the target address, if the comparison succeeds.
- The if_acmpeq, if_acmpne, if_acmplt, if_acmpge, if_acmpgt and if_acmple instructions consume two values from the evaluation stack, perform a comparison with them and branch to the target address, if the comparison succeeds.
- The if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt and if_icmple instructions consume two values from the evaluation stack, perform a comparison with them and branch to the target address, if the comparison succeeds.
- The monitorenter and monitorexit instructions allow accessing the monitor associated with each object. monitorenter increments the counter of the monitor, if it is either zero or the current thread is the same as the owner thread of the monitor. If the counter is greater than 0 and it is owned by a different thread, the current thread blocks until the counter reaches 0. monitorexit is used to decrement the counter. If it reaches 0 ownership of the monitor is reset.
- The athrow instruction throws an object of type Throwable from the evaluation stack.
- The wide prefix is used to enable four-byte indices instead of two-byte indices on some instructions: iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, or ret.
- A nop instruction, which has no effect and may be used for padding.

For a detailed description, please refer to https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

3.3 Cacao JVM

The following summarizes the key points of CACAO's architecture, focusing on the JIT compiler. The CACAO JVM implements its own class loader, which supports eager and lazy loading of classes [22]. The garbage collector uses the *Boehm GC*. CACAO currently features two JIT compiler implementations: the stage 1 compiler and an optimizing compiler.

The JIT pipeline comprises the following steps:

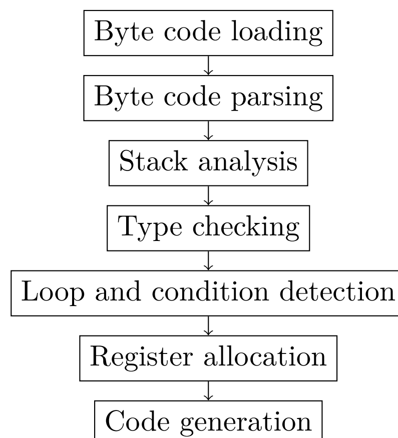


Figure 3.3: A summary of the steps performed by CACAO.

1. Loading and parsing the byte-code, which performs the following sub-steps:
   - Parses the raw Java byte-code to IR instructions.
   - Simplifies instructions: For example, instructions that load constants are unified: `ICONST_N` is mapped to `ICONST` and the immediate value is stored in the `instruction` structure. Similarly `LDC` instructions are decoded and mapped to their typed equivalent, (i.e., `ACONST`, `DCONST`, `FCONST`, `ICONST` and `LCONST`) and the value from the constant pool is stored in the IR instruction.
   - Detects basic blocks: Each block ends in a control-flow related instruction.
2. Stack-analysis performs the following tasks:
   - Checks the IR instructions for operand stack underflow and overflow, matching stack-depth and types at merging points and other type-checks.
   - Replaces specialized stack handling instructions (such as the various forms of `POP` and `DUP`) by `COPY` and `MOVE` instructions, which capture the data-flow and are better suited for register allocation.
   - Applies simple optimizations, such as inlining of constants into arithmetic instructions, such as `ICONST` and `IADD` are combined into `IADDCONST`.
3. Performs type-checking and creates a CFG (control-flow graph) of all basic blocks.
4. Applies some loop and condition optimizations.
5. Simple register allocation
6. Code generation: A simple loop over all basic blocks and instructions, which calls the architecture-specific emit functions.

3.4 Tree-parsing, `(i)burg` and `bfe`

Tree-parsing is a subset of BURS (bottom-up rewrite system) theory, which was developed by Pelegri-Llopart et al. [20]. It can be used to solve the problem of translating an expression tree (or IR term) to the best possible machine code sequence, i. e., instruction selection. It is described by Aho et al. [2, p. 493] as a set of rules, where each rule consists of a *replacement* node, an expression tree, an *action* that is a code fragment, and costs that are associated with the execution of the code fragment. Aho et al. call this a “tree-translation scheme”, however, in this work it will be called tree-parsing grammar.

Before we jump into an example (see section 3.4.3) in detail, we need to introduce some further concepts: A tree grammar is a context-free grammar, defined as $G = (N, T, S, R)$. N is the set of *non-terminals*, T is the set of *terminals*, S is the starting/root *non-terminal* and R is the set of rules, which have the following structure: $p : t$, where $p \in N$ is a non-terminal and t is a tree-pattern, an (ordered) binary tree, where leaf nodes are either non-terminal or terminal symbols and interior nodes are terminal symbols.

Tree-pattern matching as described by Aho et al. consists of two passes: The *labelling* and the *reduction* pass. In the labelling pass, the tree is traversed depth-first and the rules are matched against the sub-trees. At each node the costs associated with the rules are used to determine the best matching rule, then the sub-tree is replaced with the replacement symbol. The reduction pass is another depth-first traversal of the tree. At each node the actions associated with the matched rule (as calculated in the first pass) are executed.

The trees used in the rules partially match the input IR expression trees and the actions describe the code that is generated for each rule. The replacement nodes can be viewed as storage classes[5, p. 433] and represent the result/output of each rule.

3.4.1 `burg` and `iburg`

`burg` and `iburg` are tools that generate labellers used in tree-pattern matchers from tree-parsing grammars. Listing 3.1 shows a simple example of a tree-parsing grammar in `(i)burg` syntax. It consists of the following parts:

1. At the top, enclosed by `%{` and `%}`, there is a block of user code, that will be inserted at the top of the output.
2. Definitions of symbols: `%start` denotes the non-terminal start symbol of every tree. `%term` is followed by one or more mappings of terminal symbol (= operator) names to input symbol numbers. Terminal symbols can either be binary, unary or nullary; the arity is inferred from the usage in the grammar.
3. Separated by `%%` follows the list of rules: each rule uses the syntax `replacement-symbol ":" tree "=" rule-number "(" cost ")" ";"`, where the replacement symbol is a non-terminal symbol, (expression) trees involve any number

of terminal and non-terminal symbols, rule-numbers are stored in the labelled tree for further processing by the reducer. Costs are positive integer values used to determine the best rules for a given tree.

4. After the second %% separator, the remainder of the file is copied verbatim to the end of the output.

The labeller can be invoked for a tree by simply using `burm_label(tree);`.

Listing 3.1: A simple tree-parsing grammar using (i)burg syntax.

```
%{ /**
    * block containing user code
    * inserted verbatim at the top of the output file.
    **/ %}
%start root /* Start replacement/non-terminal symbol. */
/* Definitions of terminal symbols. */
%term ICONST=1 ILOAD=2
%term INEG=3 IADD=4 ISTORE=5
%% /* Separator */
/* Replacement: tree     = rule-number (cost); */
reg:   const             = 1           (1)   ;
const: ICONST            = 2           (0)   ;
reg:   ILOAD             = 3           (0)   ;
reg:   INEG(reg)         = 4           (1)   ;
reg:   IADD(reg, reg)    = 5           (1)   ;
root:  ISTORE(reg)       = 6           (1)   ;
reg:   IADD(const, reg)  = 7           (1)   ;
reg:   IADD(reg, const)  = 8           (1)   ;
%% /* Separator */
/**
 * block containing user code
 * appended at the end of the output file.
 **/
```

3.4.2 Rules, normal form and chain rules

As previously noted, trees in rules can consist of any number of terminal and non-terminal symbols, forming a tree potentially spanning multiple levels. For example, `root: ISTORE(IADD(reg, ICONST))` would be a valid rule tree. A semantically equivalent tree can be written in *normal form* (the concept was introduced by Balachandran et al. [3, p. 131]), if, for any sub-tree containing terminal symbols, a new non-terminal symbol/rule is introduced and the sub-tree is moved to that rule.

Listing 3.2 shows the steps of such a “normalization”. In order to avoid possible duplication of rules, *chain rules* can be introduced: These rule trees only consist of a single non-terminal symbol.

Listing 3.2: Transformation of a rule into normal form and introduction of chain rules.

```
// initial input
reg: IADD(reg, reg)
reg: ICONST
root: ISTORE(IADD(reg, ICONST))

// Add new non-terminal nf0 and extract sub-tree:

reg: IADD(reg, reg)
reg: ICONST
root: ISTORE(nf0)
nf0: IADD(reg, ICONST)

// Recursively repeat for all terminal symbols,
// leading to the following final output:

reg: IADD(reg, reg)
reg: ICONST
root: ISTORE(nf0)
nf0: IADD(reg, nf1)
nf1: ICONST

// Now because there are two equal rules involving
// the symbol ICONST a chain rule can be introduced
// to avoid duplication:
// (Note that the rules involving IADD are not
// duplicates, because they use different non-terminals.)

reg: nf1 /* chain rule */
reg: IADD(reg, reg)
root: ISTORE(nf0)
nf0: IADD(reg, nf1)
nf1: ICONST
```

The usefulness of normalization and chain rules will become apparent, when we take a look at the different approaches of implementing the instruction selector in chapter 4.

3.4.3 Labeller and Reducer Example

In the context of stack-based languages a (binary) tree can be constructed from instructions (that consume 0, 1 or 2 stack slots and push 0 or 1 stack slot onto the stack as a result) by creating a node for each instruction and adding the instructions whose results are consumed as child nodes of the current instruction/node, forming an *expression tree*. An expression tree is automatically terminated, if an instruction does not push anything onto the stack as a result or if the instruction stack is empty. Special instructions like `dup` and `swap` terminate a tree as well (see section 4.1.1 for a detailed description).

For example, the instruction sequence `ILOAD ICONST IADD ISTORE` can be written using a parenthesized prefix notation as follows: `ISTORE(IADD(ILOAD, ICONST))` and then be turned into an expression tree.

In the following an example labelling and reduction pass will be presented and explained. Figure 3.4 shows a simple Java statement translated to IR, an expression tree and a parenthesized notation used in this work. Listing 3.3 shows a grammar that can be used with our example tree.

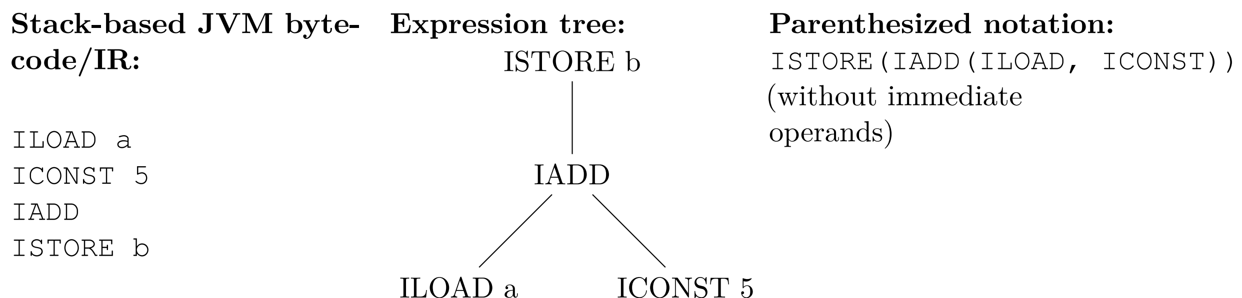


Figure 3.4: A simple Java statement `b = a + 5;` in JVM byte-code, IR tree and parenthesized notation.

The grammar in listing 3.3 uses `root` as start non-terminal symbol, which is only used in rule 6. This means that any derivable tree has the `ISTORE` symbol in its root. Rule 1 is a chain rule, connecting the `const` non-terminal with the other parts of the tree. Rules 2, 7 and 8 are specialized rules, which are used to produce better code for trees involving additions of constants. Rules 3 to 5 are normal rules allowing instructions to be nested possibly infinitely.

Listing 3.3: An example grammar with an action.

```
/* no.  rule                    # cost # action */
/* 1 */ reg:   const            # 1    # print("store const in reg\n");
/* 2 */ const: ICONST           # 0    # print("use const\n");
/* 3 */ reg:   ILOAD            # 0    # print("use reg\n");
/* 4 */ reg:   INEG(reg)        # 1    # print("negate reg\n");
```

```
/* 5 */ reg:    IADD(reg, reg)    # 1    # print("add two regs\n");
/* 6 */ root:   ISTORE(reg)       # 1    # print("store from reg\n");
/* 7 */ reg:    IADD(const, reg)  # 1    # print("add const and reg\n");
/* 8 */ reg:    IADD(reg, const)  # 1    # print("add reg and const\n");
```

The grammar only allows “statements” which assign their value to a slot in the local variable table of the JVM. “Expressions” that can be assigned are constants, values of variables, negated expressions and sums of two expressions.

Labelling Pass

As previously described, the labelling pass is a depth-first pass, working bottom-up and trying to match trees and sub-trees with rules. Tables 3.1 and 3.2 show possible reductions of the tree using different rules. The red highlight shows the sub-tree currently being processed. The blue highlight shows the replacement from the previous step. The “Rule” column shows the matching rule number and costs. In row 3 of 3.1 a chain rule is applied.

In each step the optimal rule to be used for a non-terminal is stored in the node. If more than one rule matches a given sub-tree, dynamic programming is used to determine the minimum cost cover and the rule with smallest costs is applied. The final decision, which non-terminals to use, is made at the end.

Tree	Rule	Cumulative costs
`ISTORE(IADD(ILOAD, ICONST))`	3 (cost: 0)	0
`ISTORE(IADD(reg, ICONST))`	2 (cost: 0)	0
`ISTORE(IADD(reg, const))`	1 (cost: 1)	1
`ISTORE(IADD(reg, reg))`	5 (cost: 1)	2
`ISTORE(reg)`	6 (cost: 1)	3
`root`	total	3

Table 3.1: One possible reduction for the example tree, yielding a total cost of 3.

In table 3.2 the labeller takes advantage of rule 8 and thus is able to produce a “cheaper” reduction.

Tree	Rule	Cumulative costs
`ISTORE(IADD(ILOAD, ICONST))`	3 (cost: 0)	0
`ISTORE(IADD(reg, ICONST))`	2 (cost: 0)	0
`ISTORE(IADD(reg, const))`	8 (cost: 1)	1
`ISTORE(reg)`	6 (cost: 1)	2
`root`	total	2

Table 3.2: An alternative reduction for the example tree, yielding a total cost of 2.

Reduction Pass

In the reduction pass (as implemented in listing 3.7), the tree is again visited depth first and the actions associated with the previously derived rules are executed top-down.

Interpreting the print statements as given in listing 3.3, the following listings would be produced for the two reductions presented above:

Listing 3.4: Output of the example reducer for the reduction from table 3.1.

```
use reg
use const
store const in reg
add two regs
store from reg
```

Listing 3.5: Output of the example reducer for the reduction from table 3.2.

```
use reg
use const
add reg and const
store from reg
```

The second reduction avoids one step, which leads to better code-generation as an `add` instruction taking an immediate value may be used.

3.4.4 bfe (burg front-end)

The input of `(i)burg` is deliberately kept flexible, so the labeller can be used for different applications. It does not allow the user to specify any actions associated with the rules. Also, the rule numbers need to be specified by the input. To make creating instruction selectors easier, the `bfe` script (see appendix A.1) was created at TU Wien for the Compiler-Construction Course.

Listing 3.6: A simple tree-parsing grammar using `bfe` syntax.

```
%{ /**
    * block containing user code
    * inserted verbatim at the top of the output file.
    **/ %}
%start root /* Start replacement/non-terminal symbol. */
/* Definitions of terminal symbols. */
%term ICONST=1 ILOAD=2
%term INEG=3 IADD=4 ISTORE=5
%% /* Separator */
/* tree                  # cost # action */
reg:   const             # 1    # to_reg(node);
const: ICONST            # 0    # /* nop */
```

```
reg:    ILOAD              # 0    # to_reg(node);
reg:    INEG(reg)          # 1    # ineg(node);
reg:    IADD(reg, reg)     # 1    # iadd(node);
root:   ISTORE(reg)        # 1    # istore(node);
reg:    IADD(const, reg)   # 1    # iadd_const(node);
reg:    IADD(reg, const)   # 1    # iadd_const(node);
%% /* Separator */
/**
 * block containing user code
 * appended at the end of the output file.
 **/
```

It uses a syntax similar to `iburg`; only the description of rules is simplified and extended. Each rule consists of a tree, an integer specifying the cost of the rule and an action, one executed in post-order. All of these items are separated by `#`, as can be seen in listing 3.6.

When executed `bfe` produces the following as its output: It translates all the rules to the syntax used by `iburg` and also generates a reducer, which consists of the following skeleton:

Listing 3.7: Skeleton of a reducer as produced by `bfe`.

```
void burm_reduce(NODEPTR_TYPE bnode, int goalnt)
{
  int ruleNo = burm_rule (STATE_LABEL(bnode), goalnt);
  short *nts = burm_nts[ruleNo];
  NODEPTR_TYPE kids[100];
  int i;
  /* skip error handling */
  burm_kids (bnode, ruleNo, kids);
  for (i = 0; nts[i]; i++)
    burm_reduce (kids[i], nts[i]); /* reduce kids */
  switch (ruleNo) {
    /* GENERATOR: insert a case for each rule and add the
   action as its body */
  }
}
```

The reducer can be invoked using the following call `burm_reduce(tree, 1);`. The literal `1` for the `goalnt` parameter refers to the start non-terminal symbol. So, using the previous call, the whole tree is traversed.

3.4.5 Output of **bfe** and **iburg** combined

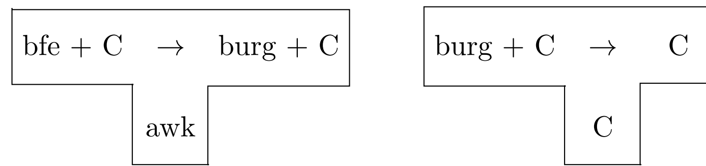


Figure 3.5: Tombstone diagrams for bfe (left) and burg (right).

Shown in figure 3.5 is the structure of both tools bfe and iburg. Note that the input of bfe consists of rules written in bfe syntax and fragments written in C. Similarly, the input of burg is written in burg syntax augmented with C code. The final result is plain C code that can be integrated into any compatible environment.

CHAPTER 4

Approaches under Examination

In the following sections four approaches for code generation and their implementation in CACAO will be discussed. We started with the "tree-parsing and dynamic programming" approach used by iburg, because it offers a working implementation of tree-parsing grammars. The code generation actions were, for the most part, taken from CACAOs stage 1 JIT compiler. Only some constant folding optimizations were removed from CACAO's stack analysis code and added as rules in the grammar to show that such optimizations can easily be performed during instruction-selection without an extra pass.

For the other approaches, "finite state automaton" and "pushdown automaton", we decided to reuse the tree-parsing grammar as understood by bfe (see section 3.4.4), because it offers a convenient way to add rules and provides all information needed for a one-pass instruction selector. This allowed us to implement a generator similar to burg, which produces a state-machine from the grammar.

For the discussion of each approach the following simplified JVM instruction set and bfe grammar will be used:

- ICONST: pushes a 32-bit constant onto the evaluation stack. Stack-effect: 0 - 1.
- ILOAD: loads the value of a 32-bit variable slot and pushes it onto the evaluation stack. Stack-effect: 0 - 1.
- INEG: pops a 32-bit value off the evaluation stack and pushes -value back onto the evaluation stack. Stack-effect: 1 - 1.
- IADD: pops two 32-bit values off the evaluation stack and pushes their sum onto the evaluation stack. Stack-effect: 2 - 1.
- ISTORE: pops a 32-bit value off the evaluation stack and stores it in a 32-bit variable slot. Stack-effect: 1 - 0.

Listing 4.1: A simple tree-parsing grammar used in this chapter.

```
%start root

reg:   const            # 1 # print("emit push_const_to_reg\n");
const: ICONST           # 0 #
reg:   ILOAD            # 0 # print("emit iload\n");
reg:   INEG(reg)        # 1 # print("emit ineg\n");
reg:   IADD(reg, reg)   # 1 # print("emit iadd\n");
root:  ISTORE(reg)      # 1 # print("emit istore\n");
reg:   IADD(const, reg) # 1 # print("emit iaddconst\n");
reg:   IADD(reg, const) # 1 # print("emit iaddconst\n");
```

4.1 Tree-parsing Automata using Dynamic Programming

This is the approach used by many code-generator generators such as BEG, Twig and (i)burg. An in-depth explanation of the labeller produced by `iburg` and the reducer produced by `bfe` can be found in section 3.4.

4.1.1 Constructing expression-trees from stack-based instruction sequences

Programs compiled to a stack-based intermediate language essentially use RPN. Because tree-parsing automata work with (expression) trees, it is necessary to convert the program from RPN into (expression) trees. We use the simple algorithm shown in listing 4.2 to convert RPN into a (binary) tree structure.

Listing 4.2: Pseudo-code showing the algorithm converting RPN into a tree structure.

```
1  eval_stack = [];
2  for instruction in code {
3      // detect instruction stack-effect
4      pop_count, push_count = instruction.stack_effect;
5      // create a new node from an instruction
6      current_node = node_from_instruction(instruction);
7      // handle stack-effect
8      switch (pop_count) {
9          case 0:
10             break;
11         case 1:
12             current_node.right = nop;
13             current_node.left = eval_stack.remove_at(0);
14             break;
15         case 2:
16             current_node.right = eval_stack.remove_at(0);
17             current_node.left = eval_stack.remove_at(0);
18             break;
19     }
20     if push_count == 1 {
21         eval_stack.insert_at(0, current_node);
22     } else {
23         // handle termination of a tree
24     }
25 }
```

Each instruction is first stored in a new node. Depending on the stack-effect of the instruction, we use 0, 1 or 2 items from the evaluation stack and store them as left and

right child node of the newly created node. In line 24 of the above algorithm, code would be added to call `iburg`.

If we assume, that there exist no instructions that consume more than 2 items and only either push 0 or 1 item, then the above algorithm is complete. The instruction sequence, that executes the statement `b = a + 5;`, i.e., `ILOAD a, ICONST 5, IADD, ISTORE b` would be translated into the following tree:

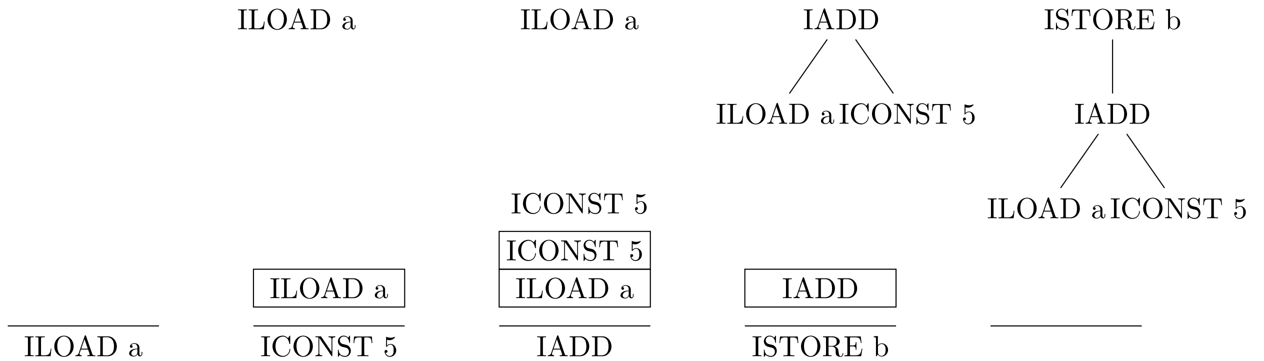


Figure 4.1: Diagram showing the evaluation stack next to the trees that are constructed by the above algorithm.

The above example also illustrates why trees are terminated once an instruction does not push any result onto the evaluation stack: When evaluating each instruction, the stack-effect is executed and instructions that push a result are used as inputs/children of the next instruction. If no result is pushed there is nothing that connects the instruction with later instructions. Because `burg` only supports binary trees, instructions that have more than two inputs, are emitted separately.

4.1.2 Integrating `iburg` in CACAO

The integration of `iburg` in CACAO operates separately on each basic block. Before calling `burm_label` and `burm_reduce` on a tree, the flat list of IR instructions provided by CACAO needs to be converted into IR expression trees. This is done by allocating tree nodes for each instruction and then constructing trees by doing symbolic evaluation of the instructions.

Almost all instructions have a known stack-effect, from which a tree can be constructed, as described in section 4.1.1. In practice the tree construction happens as follows:

Each `basicblock` has a known length (number of instructions) and a known incoming stack-height, i.e., a number of items on the evaluation stack at the start of the basic block. A stack-height > 0 commonly happens when translating conditional expressions (such as `condition ? oneValue : otherValue`) to JVM byte-code.

- For each incoming stack-item, a pseudo RESULT node is created, which can then be used in an expression tree.
- Next, the stack-effect of each instruction is calculated. Some instructions have static stack-effect (such as arithmetic instructions, loading of constants and local variables), others have dynamic stack-effect, depending on run-time information, in particular:
  - BUILTIN instructions are used to implement functionality not natively supported by the processor, such as floating point arithmetic, and also internal functions such as constructor invocations, array creation, type-checking and other special instructions. For a full list, refer to builtintable.inc in the CACAO source code. The stack-effect is derived from the target method descriptor.
  - Any of instruction doing a method invocation (i.e., INVOKESTATIC, INVOKESPECIAL, INVOKEVIRTUAL and INVOKEINTERFACE). The stack-effect is derived from the target method descriptor.
- If the calculated stack-effect has a pop_count of less or equal to 2 the number of items is consumed from the evaluation stack and if the push_count is 1, then the current instruction is pushed onto the evaluation stack.
- If the calculated pop_count is greater than 2 or push_count is 0, then the tree is terminated and the machine code is generated immediately (see also section 4.1.1).
- Note that sometimes it is necessary to keep multiple partial expression trees on the evaluation stack, so the evaluation stack is stack of expression trees. In order to avoid changing the ordering of instructions, each time a tree is emitted, all previous trees in the evaluation stack are emitted as well and replaced by RESULT nodes, which can be used again later.
- After the end of the basic block, all trees and instructions remaining on the evaluation stack are emitted. This can happen if a basic block produces a result, such as in conditional expressions.

The algorithm described above causes the JIT compiler to consume more memory than the other approaches and also tree construction and management might cause the code-generation to be slowed down. This is one of the things, we will take a look at when evaluating the performance of this approach in chapter 5.

4.2 Naïve Expansion

The current stage 1 JIT compiler implemented in CACAO uses this approach (see section 3.3 for a more detailed overview). The instructions are emitted one-by-one and in general naïve expansion has no way of generating specialized code for combinations of instructions, unless they are pre-processed in a separate pass over the instruction stream (which is what the current CACAO implementation does).

For example, the JVM byte-code given below on the left, would be naïvely emitted as four distinct assembly instructions (see column 2), because each instruction is processed individually. However, if the byte-code instructions are combined in an expression tree, it is possible to emit shorter code for this sequence, as can be seen in columns 3 and 4:

1) *Java VM byte-code*:	2) *Naïve Expansion (into IA-32 assembly language)*:	3) *Inlining constants*:	4) *Eliminate extra mov instructions*:
`ILOAD_0`	`mov 4(%esp), %ecx`	`mov 4(%esp), %ecx`	`mov 4(%esp), %eax`
`ICONST_5`	`mov $5, %edx`	`add $5, %ecx`	`add $5, %eax`
`IADD`	`add %ecx, %edx`	`mov %ecx, %eax`	
`ISTORE_1`	`mov %edx, %eax`		

The manual (naïve) expansion implementation of CACAO performs the “inlining constants” step shown above in a separate stack-analysis pass: While scanning all IR instructions in a basic block, it stops at instructions that push a constant value (such as `ICONST`) and performs a look-ahead in the instruction stream and, if the constant instruction is followed by an arithmetic or store instruction, the instructions are merged (see figure 4.2).

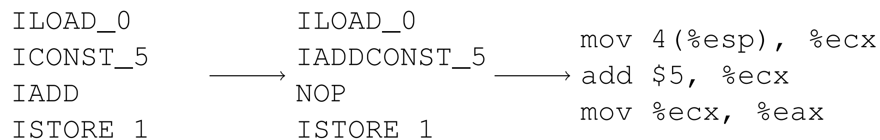


Figure 4.2: A simple constant folding transformation as done by CACAO.

Note, that in the above example the last `mov` instruction is usually omitted, if the value is used right after the calculation and the register is not needed otherwise.

4.3 Automata

In general, the automaton consumes each byte-code instruction sequentially and greedily executes code-generation action. This removes the need to build a complex tree structure – however, as code-generation actions might want to access information on previous instructions, it is necessary to maintain a stack of “active” instructions, i.e., instructions that have not been emitted.

4.3.1 Finding a suitable representation of state

The straight-forward way of representing the state of the evaluation stack after each instruction is to use the instruction op-codes directly. For example, the sequence `ILOAD a`, `ICONST 5`, `IADD`, `INEG`, `ISTORE b` (representing the statement *b = -(a + 5))*, produces the following states: `iload`, `iconst_iload`, `iadd`, `ineg` and `<empty>` as can be seen in figure 4.3.

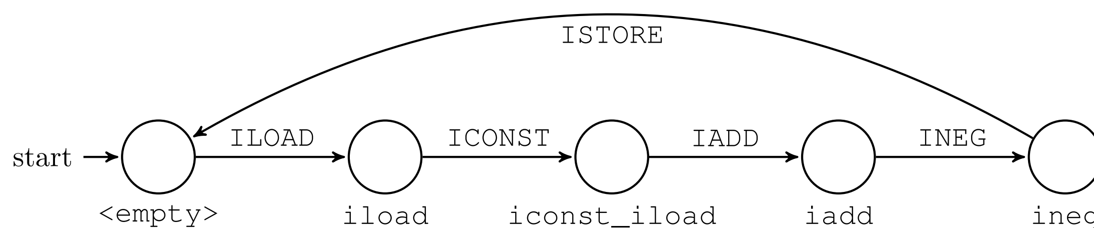


Figure 4.3: A finite-state machine modeling the sequence `ILOAD a`, `ICONST 5`, `IADD`, `INEG`, `ISTORE b` using instruction op-codes in the states.

Using the approx. 200 op-codes to uniquely identify stack state (e.g. `ICONST, ILOAD` to denote the state where a constant integer and a variable integer were previously pushed onto the stack), would produce $\sum_{i=0}^{4} 200^i \approx 1.6 * 10^9$ possible states taking stack depths up to 4 into account. However, generating a state-machine of this size would require a lot of memory and time, as the generation algorithm depends on the number of rules and the number of possible states. Each newly generated state causes the list of rules to be searched again.

Due to these problems, we were looking at other ways of representing the state of the evaluation stack. We found a possible answer in the `iburg` grammar: Each rule has a left-hand side, called `root`, `reg` and `const` in our example, which can also be used to represent the current stack state. After all, `ISTORE` does not push anything onto the stack, therefore it is mapped to `root`, `ICONST` pushes a constant (= `const`) and all other instructions work with registers (= `reg`). Note that the `const` non-terminal symbol is added to allow expressing constant optimizations in rules. Reusing the example from above, the following states would be produced: `reg`, `const_reg`, `reg` and `root`.

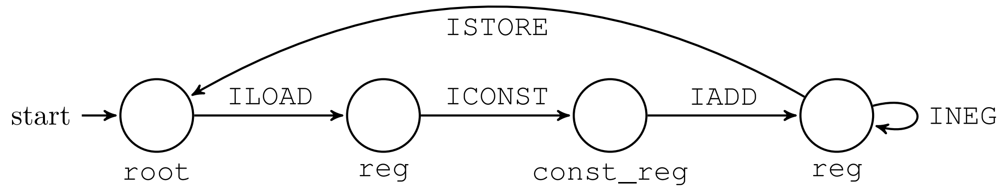


Figure 4.4: A finite-state machine modeling the sequence `ILOAD a`, `ICONST 5`, `IADD`, `INEG`, `ISTORE b` using non-terminal symbols in the states.

4.3.2 The Automaton Generator Algorithm

The basic idea is a *worklist algorithm* that - starting from a single state representing the empty stack or `root` - repeatedly applies rules to the current state. Each application produces a new (stack) state and added to the worklist. If the state is a duplicate of an existing state, the existing state is reused, and not added to the worklist. As the algorithm would continue to produce new states infinitely, there is a maximum stack-depth after which the algorithm stops to generate new states and only produces transitions to existing states.

Applying a rule to a state is in our case the same as interpreting the rule's stack-effect. For example, applying rule 3 (from listing 4.1) to the empty state produces a new state consisting of a `reg` stack-item. Applying rule 4 to that new state would not produce another new state, because `INEG` has `1 - 1` as its stack-effect.

Chain rules (rule 0 in our example) cannot be directly applied, because they produce no stack-effect and are used to declare aliases between non-terminal symbols. Looking at rule 4, when applied to the state `const`, the rule does not match the input state exactly because `reg != const`. If no exact match (as described in the previous paragraph) can be performed, the algorithm first applies chain rules and then tries again to apply the other rules.

When generating the state machine, the `action` is emitted in the transitions. For chain rules, the actions are emitted before the action of the rule with stack-effect.

See the following pseudo-code for a high-level description of the algorithm:

Listing 4.3: Pseudo-code showing the general idea of the generator algorithm.

```
def generate(terminals, nonterminals, rules)
    chain_rules, rules = SplitChainRules(rules)
    s0 = new State()
    states = [ s0 ]
    edges = []
    worklist.Enqueue(s0)
    while (worklist.count > 0)
        s = worklist.Deque()
```

```
        foreach (r in rules)
            sn = ApplyExact(r, s)
            if (sn is not null)
                sn = states.GetOrAdd(sn)
                edges.add(MakeEdge(s, sn, r))
                if (sn.StackDepth <= MAX_DEPTH)
                    worklist.Enqueue(sn)
                end
            end
        end

        foreach (r in rules)
            sn = ApplyWithChains(r, s, chain_rules, out chain)
            if (sn is not null)
                sn = states.GetOrAdd(sn)
                edges.add(MakeEdge(s, sn, r, chain))
                if (sn.StackDepth <= MAX_DEPTH)
                    worklist.Enqueue(sn)
                end
            end
        end
    end
end
```

The interface of the generated state-machine consists of the following functions:

- `int next(int current, int symbol);` This function returns the next state for the given `current` state and a `symbol`.
- `void execute_action(int state, int symbol, /* implementation-specific data */ ...);` Executes the `action` defined in the grammar. In the prototype, the implementation-specific data is not yet extracted into a replaceable data-type.

Now, we will look at a short example and the generated automaton:

Listing 4.4: Example rules

```
1 reg:   const           # 1    # to_reg(node);
2 const: ICONST          # 0    # /* nop */
3 reg:   ILOAD           # 0    # to_reg(node);
4 reg:   INEG(reg)       # 1    # ineg(node);
5 reg:   IADD(reg, reg)  # 1    # iadd(node);
```

```
6 root:  ISTORE(reg)       # 1    # istore(node);
7 reg:   IADD(const, reg)  # 1    # iadd_const(node);
8 reg:   IADD(reg, const)  # 1    # iadd_const(node);
```

Figure 4.5: A finite-state machine as produced by our generator for a maximum stack-depth of 2. It starts at the empty state at the top. Note the line/rule numbers from listing 4.4 in parentheses. Also note that some transitions have been omitted for clarity.

4.3.3 The implementation in CACAO

One core part of this thesis was the implementation of a prototype that allows an empirical evaluation of each approach. The implementation in CACAO uses the following specialized function signatures for the state-machine functions:

- `int next(int current, int symbol);`
- `void execute_action(int state, int symbol, struct jitdata *jd, struct instruction *iptr, struct instruction **stack);`

`struct jitdata *jd` provides access to the global state of the CACAO JIT. `struct instruction *iptr` points to the JVM instruction currently being processed and `struct instruction **stack` points to the other JVM instructions, whose values are on the symbolic evaluation stack.

Similar to the `iburg`-based approach, some stack-management is necessary, because rules may want to access information of previous instructions. For example, when inlining an `ICONST` into `IADD`, it is necessary for the emitting code to have access to both instructions and not just the last, which would be `IADD`.

The state maintained in the implementation consists of the following parts:

- `current`: The current state of the state-machine.
- `stack` and `tos`: List of JVM instructions previously processed but still "active".
- `current_stackheight`: The actual stack-height of the evaluation stack. This might exceed the maximum supported stack-height of the state-machine.
- `dump`: If set to `true`, the state-machine code is skipped and each JVM instruction is emitted directly.

The JVM instructions are processed one-by-one and first, the stack-effect of each JVM instruction is calculated, as described previously in section 4.1.2. If the stack-effect of the JVM instruction does not fit into the state-machine, i.e., if pop count is greater 2, or push count is 0, or the current stack-height exceeds the maximum supported height of the state-machine, or the JVM instruction has dynamic pop count, then all JVM instructions on the evaluation stack, that have not been previously emitted, are emitted eagerly followed by the current JVM instruction. If the evaluation stack-height is not empty after this, it continues to directly emit ("dump") further JVM instructions eagerly, until the evaluation stack is empty and the state-machine is reset. Otherwise the current state and the current JVM instruction are fed into the state-machine and all associated actions are executed. If the end of the basic block is reached, all remaining JVM instructions are emitted eagerly.

Compared to the iburg-based approach it uses less memory, because there is no need to allocate a node struct for every JVM instruction. As the generated state-machine has a maximum stack-height it supports, only that many JVM instructions need to be kept in memory. Also, because there is no need to construct tree structures, the implementation can be greatly simplified.

4.3.4 Comparison with the manual approach

Compared with the manual approach, the finite-state machine should be more performant, because it can avoid multiple passes over the instruction sequence and only needs to calculate the next state from the current state and the current instruction, which essentially is one lookup.

Another advantage of using a grammar-based approach over a hand-written code generator is that optimizations can be derived from the grammar and are more systematic.

4.3.5 Comparison with **gburg**

One important difference is that the automaton generated by our generator produces states that resemble actual stack states that are produced during runtime. gburg on the other hand, implements each non-terminal as a state.

While using the same basic idea of using storage classes/non-terminal symbols of the tree-parsing grammar to model the different states of the state-machine, the solution described in this work does not require a separate "constructive grammar" to do its work. Our generator is intentionally kept simple and primarily relies on the author of the tree-parsing grammar to make sure to not include any conflicting rules.

Similar to gburg, the generator is unable to properly handle two rules, which use the same terminal symbol with different left-hand side non-terminals, such as rules 1 and 2 in listing 4.5.

Listing 4.5: Pseudo-Code example for a conflict requiring look-ahead.

```
reg: IADD(reg, reg)
addr: IADD(reg, const)
reg: LOADADDR(addr)
reg: const
reg: addr
```

Because const can be promoted to reg both rules 1 and 2 are applicable. This is solved by our generator by trying to match input non-terminals exactly, only if that fails, chain rules are considered by the generator. However, using the greedy approach by our generator (and also by gburg), rules that differ in their replacement symbol cannot be fully distinguished and our generator would produce only one transition that implements the rule that matches the inputs exactly, without applying any chain rules.

4.4 Pushdown Automata

While working on the finite-state machine and tree-parsing approach, it became apparent that the main bottle-neck with these approaches is stack-management and tree-construction. A pushdown automaton is a finite-state machine with a built-in stack, which would not provide more flexibility or a way to solve these problems efficiently, hence this approach was not investigated further.

CHAPTER 5

Results

In the following chapter, the measurements and comparison result are presented and discussed.

The following measurements are made:

- Compiler execution time: The benchmarks are executed with CACAO's `-enable-rt-timing` compilation flag set. This flag turns on performance measurement and logging in CACAO and produces a `rt-timing.log` file, which contains information about the time consumed by the different phases of the CACAO JIT compiler.
- Instruction counts and rule hit counts: All class files of the GNU class path 0.99 are compiled and the absolute counts of total instructions and applied rules are recorded.
- Reset hit counts: The benchmarks are executed with logging turned on and the number of instructions processed in total and number of instructions causing the state-machine to be reset are recorded.
- Generated code size: The benchmarks are executed with logging turned on and the size of the generated code are recorded.
- Total execution time: The benchmarks are executed and the total execution time is measured using the Linux `perf stat` command.

5.1 Compiler execution time

We measured the compilation time of three simple Java programs. The source code of these can be found in appendix A.2. See figure 5.1 for a comparison of the time spent in the `codegen` phase. The manual approach was the fastest for all test-cases, the `iburg`

approach was around 2x slower in all test-cases and the finite-state machine spent around 1.5x more time in the `codegen` phase compared to the manual approach.

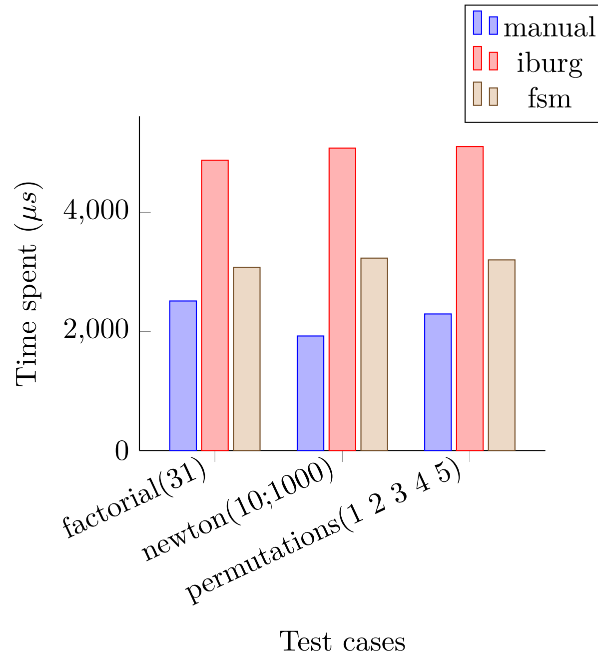


Figure 5.1: Bar chart showing the average time (in μs) spent in the codegen phase of simple test cases.

5.2 Generated code size

The size of the generated code is another possible metric to measure the performance of a compiler. Our results indicate that the code generated by our implementations is generally larger as the code generated by the manual approach. For this metric, we again used all classes of the GNU classpath 0.99. Note that the numbers in figure 5.2 also contain padding at the start or end of the methods.

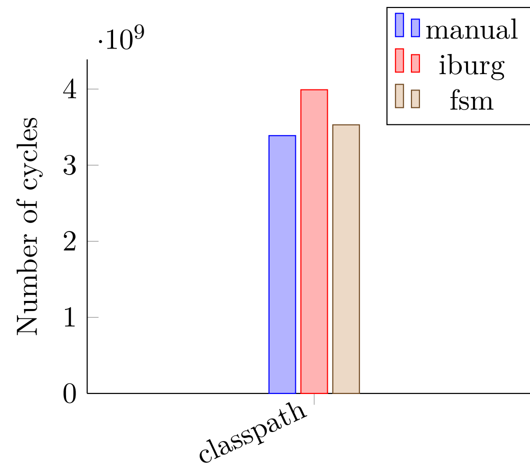


Figure 5.3: Bar chart showing the average number of cycles spent when compiling the GNU 0.99 classpath.

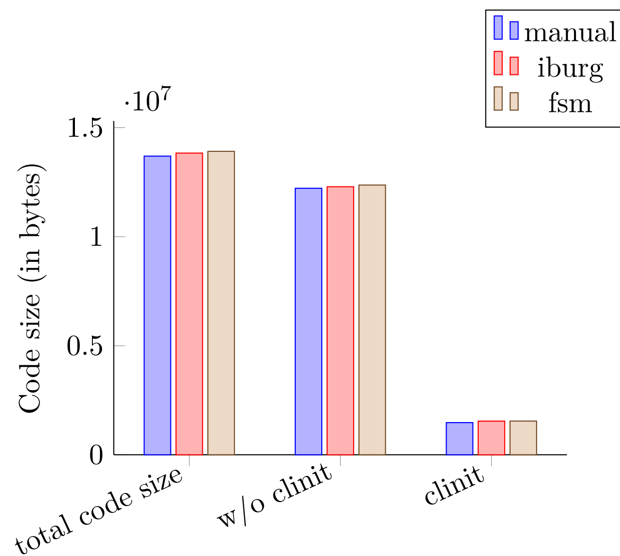


Figure 5.2: Bar chart showing the generated code size of the GNU classpath 0.99.

5.3 Total execution time

Finally, we also measured the number of cycles needed to run a few test cases. The first test case was measuring the execution of `cacao -cp path/to/classes -XX:+CompileAll`. As can be seen in figure 5.3, the difference between the manual and iburg approaches is quite large, while the finite-state machine approach is pretty close, but still not as fast as the manual approach.

Next, we again measured the execution of three simple Java programs. The source code of these can be found in appendix A.2. The results of the measurements can be seen in figure 5.4. The test cases were repeated 10 times and the values shown below are the average. Again, the manual approach is the fastest, closely followed by the finite-state machine approach.

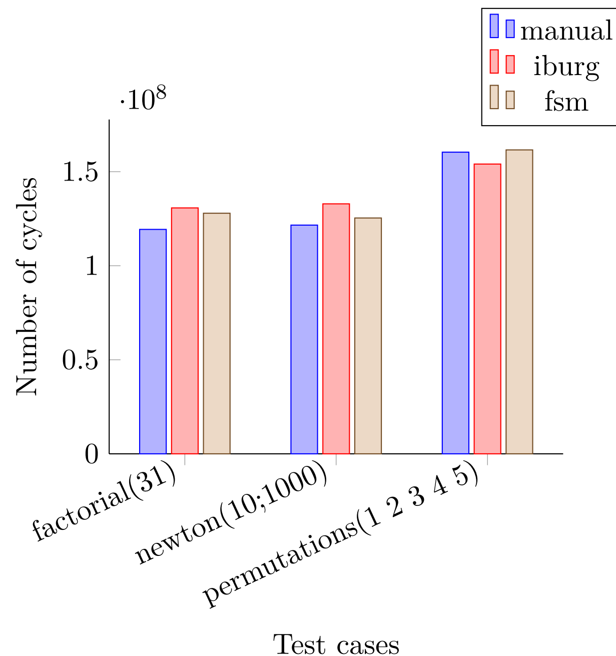


Figure 5.4: Bar chart showing the average number of cycles spent per run of simple test cases.

Of course, the relatively small size of the benchmarks makes the results not very representative. The numbers presented in chapter 5.2 are probably more meaningful.

5.4 Instruction counts and applied rules

The GNU class path 0.99 as used by CACAO contains 7346 classes and its methods contain 1,481,849 instructions in total. The numbers shown in figure 5.5 describe the number of times a constant-inlining optimization was performed in the manual, tree-parsing and finite-state machine approaches respectively. The first column for each approach shows total count of rule applications. The next two columns are the number of rule applications in JVM class initializers and for all other methods.

Note that some instructions have not been included in the diagram: `AASTORE`, `BASTORE`, `CASTORE`, `LASTORE` and `SASTORE`. These instructions consume three inputs from the evaluation stack and this is something which does not fit as well into the binary tree structure used by the `iburg` tree grammar. There are ways to model these instructions in

binary trees, however, as in our sample the majority of the instructions are only found in class initializers (where they presumably are used for one-time static array initialization). See table 5.1 for a comparison.

	total	clinit only	w/o clinit	clinit only %
AASTORECONST	410	332	78	80.98%
BASTORECONST	1812	1502	309	82.95%
CASTORECONST	12,694	12,635	59	99.54%
IASTORECONST	5983	5499	484	91.91%
LASTORECONST	1188	1184	4	99.66%
SASTORECONST	3053	3014	39	98.72%

Table 5.1: Constant array store instruction optimizations and their uses.

Looking at the diagram in figure 5.5 it becomes very clear, that compared to the manual approach, the finite-state-machine is unable to detect as many optimizations as one would expect. The total number of optimizations applied in the manual and finite-state machine implementations is 49,301 vs. 7446, which is only about 15%. If class initializers are excluded, the numbers are 23,585 vs. 7395, which is 31%. The tree-parsing approach is almost as "good" as the manual approach, which at least validates the implementation.

These results made us wonder why the finite-state machine approach is worse than the manual and tree-parsing approach. An important difference between these two and the finite-state machine approach is that for some instructions the state-machine cannot be used and must be reset. In section 5.5 we take a look at the different reasons for these "resets".

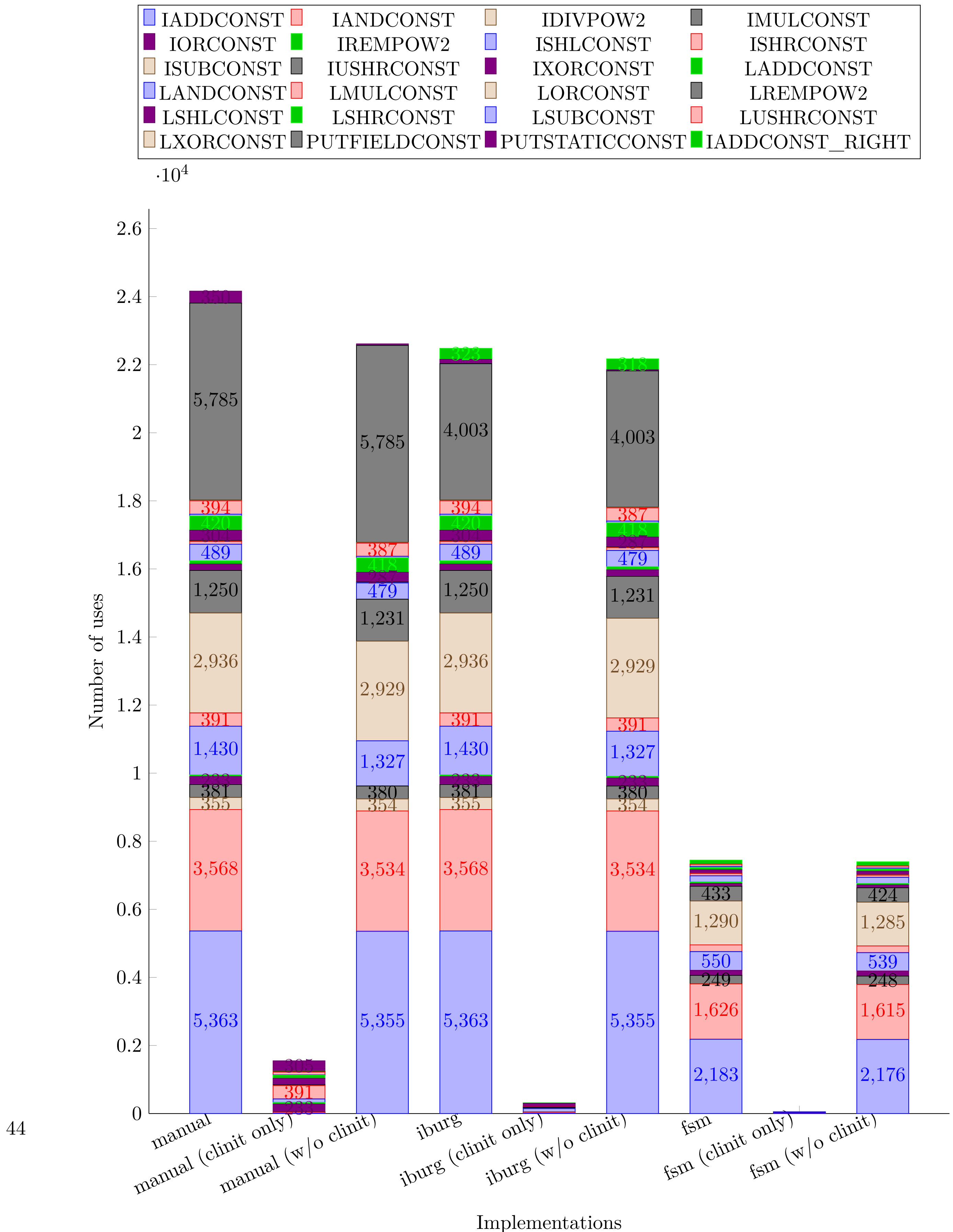


Figure 5.5: Diagram showing the absolute numbers of rule applications for the three different implementations.

5.5 Reset hit counts

As previously described, the finite-state machine cannot be used in the following cases:

- The height of the evaluation stack exceeds the height supported by the state-machine. In our case we generated a state-machine that supports heights up to 4 items.
- The instruction processes more than two items from the evaluation stack or has a dynamic stack-effect. For example, array-based store instructions or method invocation instructions.
- Some instructions are special and have no effect on the evaluation stack or are not fully supported: `GOTO`, `JSR`, `IINC`, `COPY` and `MOVE`. The first two instructions are control-flow instructions, which do not properly operate on the evaluation stack. `GOTO` has no stack-effect at all and `JSR` modifies the stack, but still does not fit well into the expression-tree concept. Similarly, `IINC` only operates on a local variable slot and not on the evaluation stack. `COPY` and `MOVE` are instructions that are used by CACAO to model data-flow of the evaluation stack in register allocation, when stack-manipulation instructions such as `POP` and `SWAP` are used.
- When an instruction that only consumes items from the stack, but does not push a result, a reset happens as well.

In our tests using the GNU class path 0.99 a reset condition occurred 130,615 times. In the following, we will take an in-depth look at the different reasons. In figure 5.6 the column “Other” shows the remaining resets, which we could not assign to a specific group and we did not look into them further.

5.5.1 Arity mismatch

If we divide the resets resulting from arity mismatches further and take a look how often a specific arity of an instruction causes a reset, we see that arities up to 3 are responsible for most of the resets (see figure 5.7).

When looking at the JVM language and our tree grammar, we can see that there are some instructions accepting an arbitrary number of arguments:

- `BUILTIN`
- `INVOKESPECIAL`
- `INVOKEINTERFACE`
- `INVOKESTATIC`

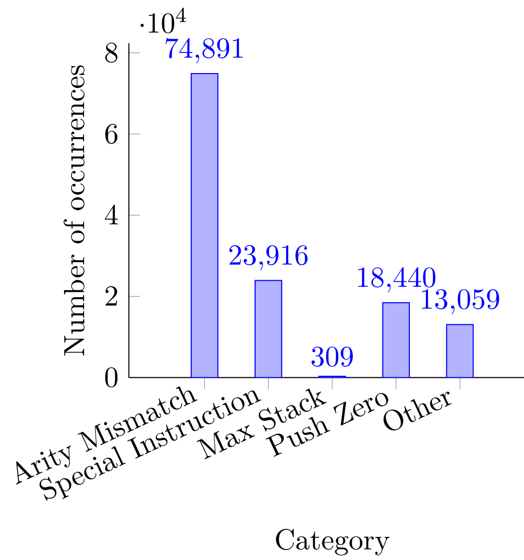


Figure 5.6: Histogram showing the general distribution of causes for resets.

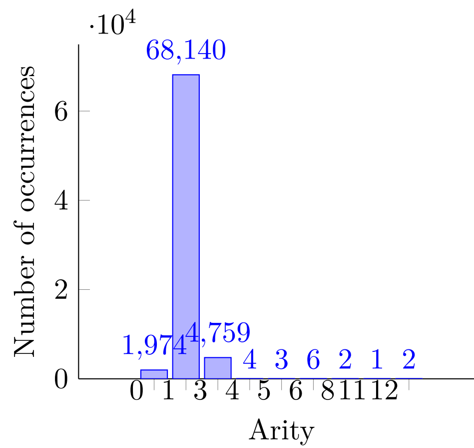


Figure 5.7: Histogram showing the number of arity mismatches per arity.

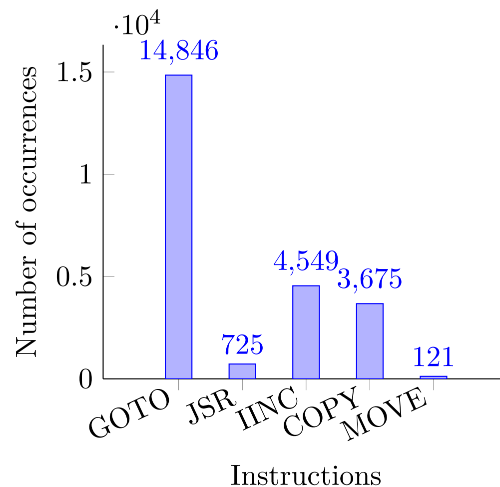


Figure 5.8: Histogram showing the number of special instructions causing a reset.

- `INVOKEVIRTUAL`
- `INVOKEDYNAMIC`
- `MULTIANEWARRAY`

In addition to that, the `*ASTORE` group of instructions consumes three arguments from the evaluation stack.

All these instructions are modeled in our grammar as accepting either 0 (for `*ASTORE`) or 2 arguments for all other special cases. In order to solve this problem (at least for less than 3 arguments) a possible solution is to introduce separate symbols in the grammar such as `BUILTIN_0`, `BUILTIN1_1` and `BUILTIN_2` so that every arity is easily distinguishable in the state-machines.

We did not implement this in our prototype, because the fix would only reduce the number of resets, but it would not improve code-quality overall, because no there currently exists no optimization rule that works with any of the above instructions.

5.5.2 Special instructions

The control-flow instructions `GOTO` and `JSR` make up the majority of special instructions causing a reset (see figure 5.8). Because these instructions do not have a normal stack-effect but rather cause transfer of control, it is important that all instructions prior to these are emitted in the current basic block.

The other JVM instruction that does not directly operate on the evaluation stack but has side-effects is the `IINC` instruction.

The `COPY` and `MOVE` instructions do not have a proper stack-effect as well and therefore don't fit in the state-machine concept either.

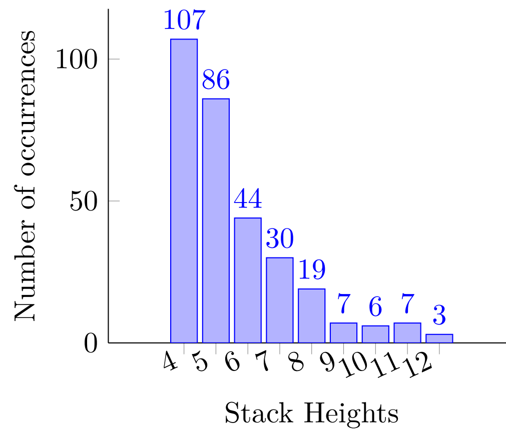


Figure 5.9: Histogram showing the stack heights causing a reset.

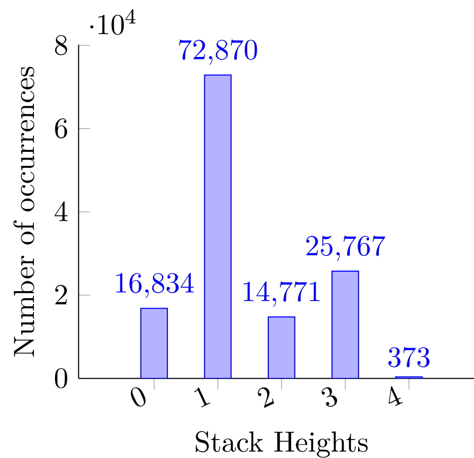


Figure 5.10: Histogram showing the stack heights before a reset.

5.5.3 Maximum stack

Figure 5.9 shows the distribution of stack heights that trigger a reset: Because the generated state-machine only supports stack-heights up to 4 items, it cannot be used if an instruction operates at that limit.

Increasing the maximum supported stack-height in the state-machine is possible, but compared to other causes for a state-machine reset, the number is quite small.

5.5.4 Reset heights

Another interesting metric is the stack height of the state-machine right before a reset happens. As can be seen in figure 5.10 most of the time a reset happens with one item on the stack.

5.5.5 Reset run length and state-machine run length

There are two more attributes that can be used to understand the performance and quality of the state-machine are “reset runs” and “state-machine runs”. A longer state-machine run means that more instructions have been processed without requiring a reset, whereas the length of “reset runs” shows how many instructions are needed for the code feeding the state-machine to return to using the state-machine.

The shortest reset run is one instruction and the longest is 10,466 instructions. The average reset run is 6.3 instructions. The shortest finite-state machine run is zero instructions and longest is 329 instructions. The average finite state.machine run is 2.45 instructions.

Comparing these numbers to the median and average length of a basic block, which is 5 and 7.85 instructions, we can deduce that on average a basic block cannot be completely processed by the finite-state machine.

CHAPTER 6

Further Work

During the writing of this thesis the following ideas came up, but investigating them was beyond the scope of this work.

- For simplicity, we injected our code generator in the `codegen` phase of CACAO. However, usually instruction selection is performed at an earlier stage in the compilation pipeline before register allocation takes place. It would be possible to reduce the number of registers allocated by performing instruction selection in advance.

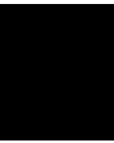
- For the `iburg` approach we have seen that the creation of expression trees is expensive. Switching the whole compilation pipeline to a tree-based IR, would improve performance and reduce memory usage, because it would remove the need for extra allocations.

- It might be possible to reduce the size of the generated automaton by introducing a separate table of function pointers, which can be addressed much faster than the generated switch tables.

- When comparing the iburg and fsm implementations further research is necessary to find out, why exactly there is a difference in the "applied rule counts". In theory both approaches should lead to similar results for the grammar used with CACAO, because there are no complex rules.

- JVM instructions that work with exactly three arguments could be implemented in a binary tree using an additional node: For example, `AASTORE(address, index, value)` could be represented as binary tree as `AASTORE(_ADDRESS(address, index), value)`.

- The BFE grammar could be extended to support more non-terminal symbols and other rules.

CHAPTER 7 

Conclusion

This work demonstrates that it is possible to generate an efficient finite-state automata for stack-based languages from tree-parsing grammars.

The results presented in the previous chapters demonstrate that the use of finite-state automata as a means of instruction selection/code generation is a more efficient strategy than using tree-parsing algorithms that require two passes.

The code-size produced by the finite-state automaton is comparable to CACAO's hand-written code-generator. One advantage of the finite-state automata approach is that rules can be written in a declarative fashion and there is no need to manually perform pattern-matching. Of course, due to the various assumptions made to simplify the problem, it is not as expressive as the `iburg` approach.

For projects that want to use a code generator generator to solve the problem of instruction selection, the finite-state machine approach and generator presented in this work are a viable alternative to `iburg`, especially if the problems mentioned in the previous chapters are solved.

Bibliography

[1] A. V. Aho and S. C. Johnson. “Optimal Code Generation for Expression Trees”. In: *J. ACM* 23.3 (July 1976), pp. 488–501. ISSN: 0004-5411. DOI: 10.1145/321958.321970. URL: https://doi.org/10.1145/321958.321970.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”. In: *ACM Trans. Program. Lang. Syst.* 11.4 (Oct. 1989), pp. 491–516. ISSN: 0164-0925. DOI: 10.1145/69558.75700. URL: https://doi.org/10.1145/69558.75700.

[3] A. Balachandran, D. M. Dhamdhere, and S. Biswas. “Efficient Retargetable Code Generation Using Bottom-up Tree Pattern Matching”. In: *Comput. Lang.* 15.3 (June 1990), pp. 127–140. ISSN: 0096-0551. DOI: 10.1016/0096-0551(90)90006-B. URL: https://doi.org/10.1016/0096-0551(90)90006-B.

[4] Gabriel Hjort Blindell. *Instruction Selection - Principles, Methods, and Applications.* Springer, 2016. ISBN: 978-3-319-34017-3; 978-3-319-34019-7.

[5] Dmitri Boulytchev. “BURS-Based Instruction Set Selection”. In: *Perspectives of Systems Informatics.* Ed. by Irina Virbitskaite and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 431–437. ISBN: 978-3-540-70881-0.

[6] John Bruno and Ravi Sethi. “Code Generation for a One-Register Machine”. In: *J. ACM* 23.3 (July 1976), pp. 502–510. ISSN: 0004-5411. DOI: 10.1145/321958.321971. URL: https://doi.org/10.1145/321958.321971.

[7] B. E. Carpenter and R. W. Doran, eds. *A. M. Turing’s ACE Report of 1946 and Other Papers.* Vol. 10. Charles Babbage Institute Reprint Series for the History of Computing. Cambridge, MA, USA: MIT Press, 1986, pp. vii+141. ISBN: 0-262-03114-0.

[8] David R. Chase. “An Improvement To Bottom-up Tree Pattern Matching”. In: *Fourteenth Annual ACM Symposium on Principles of Programming Languages.* 1987, pp. 168–177.

[9] H. Emmelmann, F.-W. Schröer, and Rudolf Landwehr. “BEG: A Generator for Efficient Back Ends”. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. PLDI ’89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 227–237. ISBN: 089791306X. DOI: 10.1145/73141.74838. URL: https://doi.org/10.1145/73141.74838.

[10] M. Anton Ertl, Kevin Casey, and David Gregg. “Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata”. In: *SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’06)*. 2006, pp. 52–60. ISBN: 1-59593-320-4. URL: http://www.complang.tuwien.ac.at/papers/ertl+06pldi.ps.gz.

[11] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805316701.

[12] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. “Engineering a Simple, Efficient Code-Generator Generator”. In: *ACM Lett. Program. Lang. Syst.* 1.3 (Sept. 1992), pp. 213–226. ISSN: 1057-4514. DOI: 10.1145/151640.151642. URL: https://doi.org/10.1145/151640.151642.

[13] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. “BURG: Fast Optimal Instruction Selection and Tree Parsing”. In: *SIGPLAN Not.* 27.4 (Apr. 1992), pp. 68–76. ISSN: 0362-1340. DOI: 10.1145/131080.131089. URL: https://doi.org/10.1145/131080.131089.

[14] Christopher W. Fraser and Todd A. Proebsting. “Finite-State Code Generation”. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. PLDI ’99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 270–280. ISBN: 1581130945. DOI: 10.1145/301618.301680. URL: https://doi.org/10.1145/301618.301680.

[15] K. John Gough. “Bottom-up Tree Rewriting Tool MBURG”. In: *SIGPLAN Not.* 31.1 (Jan. 1996), pp. 28–31. ISSN: 0362-1340. DOI: 10.1145/249094.249110. URL: https://doi.org/10.1145/249094.249110.

[16] David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. “The case for virtual register machines”. In: *Science of Computer Programming* 57.3 (2005). Advances in Interpreters, Virtual Machines and Emulators, pp. 319–338. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2004.08.005. URL: https://www.sciencedirect.com/science/article/pii/S0167642305000389.

[17] Roberto Ierusalimschy, Luiz Figueiredo, and Waldemar Celes. “The Implementation of Lua 5.0”. In: *J. UCS* 11 (Jan. 2005), pp. 1159–1176.

[18] D. C. Ince, ed. *Collected Works of A. M. Turing: Mechanical Intelligence*. Amsterdam, The Netherlands: North-Holland, 1992, pp. xix+227. ISBN: 0-444-88058-5.

[19] Dan C. Marinescu. “Chapter 10 - Cloud Resource Virtualization”. In: *Cloud Computing (Second Edition)*. Ed. by Dan C. Marinescu. Second Edition. Morgan Kaufmann, 2018, pp. 365–402. ISBN: 978-0-12-812810-7. DOI: https://doi.org/10.1016/B978-0-12-812810-7.00013-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128128107000133.

[20] Eduardo Pelegrí-Llopart and Susan L. Graham. “Optimal Code Generation for Expression Trees: An Application of the BURS Theory”. In: *Fifteenth Annual ACM Symposium on Principles of Programming Languages*. 1988, pp. 294–308.

[21] Todd A. Proebsting and Benjamin R. Whaley. “One-Pass, Optimal Tree Parsing - With Or Without Trees”. In: *Proceedings of the 6th International Conference on Compiler Construction*. CC ’96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 294–308. ISBN: 3540610537.

[22] Christian Thalinger. “Optimizing and Porting the CACAO JVM”. MA thesis. 2004.

[23] A. M. Turing. *Proposal for Development in the Mathematics Division of an Automatic Computing Engine (ACE)*. Report E.882, Executive Committee. Reprinted in [7, pp. 20-105] and [18, pp. 1–86]. Teddington, UK: National Physical Laboratory, 1945.

List of Figures

3.1 Diagram of a stack, showing push and pop operations. Each step shows the stack before the operation, above the next operation. 7
3.2 A simplified overview of the JVM architecture. It also shows that Java is compiled first into JVM byte-code using the Java compiler `javac`. 10
3.3 A summary of the steps performed by CACAO. 15
3.4 A simple Java statement `b = a + 5;` in JVM byte-code, IR tree and parenthesized notation. 19
3.5 Tombstone diagrams for `bfe` (left) and `burg` (right). 23

4.1 Diagram showing the evaluation stack next to the trees that are constructed by the above algorithm. 28
4.2 A simple constant folding transformation as done by CACAO. 30
4.3 A finite-state machine modeling the sequence `ILOAD a`, `ICONST 5`, `IADD`, `INEG`, `ISTORE b` using instruction op-codes in the states. 31
4.4 A finite-state machine modeling the sequence `ILOAD a`, `ICONST 5`, `IADD`, `INEG`, `ISTORE b` using non-terminal symbols in the states. 32
4.5 A finite-state machine as produced by our generator for a maximum stack-depth of 2. It starts at the empty state at the top. Note the line/rule numbers from listing 4.4 in parentheses. Also note that some transitions have been omitted for clarity. 34

5.1 Bar chart showing the average time (in μs) spent in the codegen phase of simple test cases. 40
5.3 Bar chart showing the average number of cycles spent when compiling the GNU 0.99 classpath. 41
5.2 Bar chart showing the generated code size of the GNU classpath 0.99. . . . 41
5.4 Bar chart showing the average number of cycles spent per run of simple test cases. 42
5.5 Diagram showing the absolute numbers of rule applications for the three different implementations. 44
5.6 Histogram showing the general distribution of causes for resets. 46
5.7 Histogram showing the number of arity mismatches per arity. 46
5.8 Histogram showing the number of special instructions causing a reset. . . 47
5.9 Histogram showing the stack heights causing a reset. 48

5.10 Histogram showing the stack heights before a reset. 48

List of Tables

3.1 One possible reduction for the example tree, yielding a total cost of 3. . . 20
3.2 An alternative reduction for the example tree, yielding a total cost of 2. . 20

5.1 Constant array store instruction optimizations and their uses. 43

Listings

3.1 A simple tree-parsing grammar using `(i)burg` syntax. 17
3.2 Transformation of a rule into normal form and introduction of chain rules. 18
3.3 An example grammar with an action. 19
3.4 Output of the example reducer for the reduction from table 3.1. 21
3.5 Output of the example reducer for the reduction from table 3.2. 21
3.6 A simple tree-parsing grammar using `bfe` syntax. 21
3.7 Skeleton of a reducer as produced by `bfe`. 22
4.1 A simple tree-parsing grammar used in this chapter. 26
4.2 Pseudo-code showing the algorithm converting RPN into a tree structure. 27
4.3 Pseudo-code showing the general idea of the generator algorithm. . . . 32
4.4 Example rules 33
4.5 Pseudo-Code example for a conflict requiring look-ahead. 36
Factorial.java 65
Permutations.java 65
NewtonMethod.java 66

Appendix

A.1 **awk** script used to generate the reducer

The following grammar and script was used to produce a reducer for `iburg`:

```
1  #!/bin/nawk -f
2
3  # AWK script for generating a reducer for iburg
4  # this is a modified version of the BFE script
5
6  function ltrim(s) { sub(/^[ \t\r\n]+/, "", s); return s }
7  function rtrim(s) { sub(/[ \t\r\n]+$/, "", s); return s }
8  function trim(s) { return rtrim(ltrim(s)); }
9
10 /^%%$/ {
11   part+=1;
12   FS="#";
13   print $0;
14   rule=1;
15   next;
16 }
17 part==0
18 part==1 && NF>0 && /^\s*[^\/]{2}/ {
19   printf "%s = %d",$1,rule;
20   if ($2!="")
21     printf " (%s)",$2;
22   printf ";\n";
23   action[rule]=trim($3);
24   rule++;
25   next;
26 }
27 part==2
28 END {
29   if (part==1)
30     print"%%"
```

```
31   print"void burm_reduce(NODEPTR_TYPE bnode, int goalnt)";
32   print"{";
33   print"  int ruleNo = burm_rule (STATE_LABEL(bnode), goalnt)
      ;";
34   print"  short *nts = burm_nts[ruleNo];";
35   print"  NODEPTR_TYPE kids[100];";
36   print"  int i;";
37   print"";
38   print"#if DEBUG";
39   print"  fprintf (stderr, \"%s %d\\n\", burm_opname[bnode->op],
       ruleNo);  /* display rule */";
40   print"  fflush(stderr);";
41   print"#endif";
42   print"  if (ruleNo==0) {";
43   print"    fprintf(stderr, \"Tree cannot be derived from start
       symbol\\n\");";
44   print"    exit(1);";
45   print"  }";
46   print"  burm_kids (bnode, ruleNo, kids);";
47   print"  for (i = 0; nts[i]; i++)";
48   print"    burm_reduce (kids[i], nts[i]);    /* reduce kids
      */";
49   print"";
50   print"";
51   print"  switch (ruleNo) {";
52   for (i in action) {
53     if (action[i]!="") {
54       print "  case "i":";
55       print "    "action[i];
56       print "    break;";
57     }
58   }
59   print"  }";
60   print"}";
61 }
```

The generated reducer retrieves the assigned rule number from each node, visits all child nodes and then executes the semantic action.

A.2 Source code of custom Java benchmarks

A.2.1 Factorial

The `Factorial` program implements $n!$ using a recursive algorithm.

```
1  class Factorial {
2      public static void main(String[] args) {
3          final int n = Integer.parseInt(args[0]);
4          System.out.println(fact(n));
5      }
6
7      static int fact(int n) {
8          if (n < 2)
9              return 1;
10         return n * fact(n - 1);
11     }
12 }
```

A.2.2 Permutations

The `Permutations` program implements calculating all permutations of an array of integers.

```
1  import java.util.*;
2
3  class Permutations {
4      static List<int[]> permutations(int[] nums) {
5          List<int[]> result = new ArrayList<>();
6          permutationsHelper(nums, 0, result);
7          return result;
8      }
9
10     static void permutationsHelper(int[] nums, int start, List<int[]> result) {
11         if (start == nums.length) {
12             result.add(nums.clone());
13             return;
14         }
15         for (int i = start; i < nums.length; i++) {
16             swap(nums, start, i);
17             permutationsHelper(nums, start + 1, result);
18             swap(nums, start, i);
19         }
20     }
21
```

```
22      static void swap(int[] nums, int i, int j) {
23          int temp = nums[i];
24          nums[i] = nums[j];
25          nums[j] = temp;
26      }
27
28      public static void main(String[] args) {
29          int[] numbers = new int[args.length];
30          for (int i = 0; i < numbers.length; i++) {
31              numbers[i] = Integer.parseInt(args[i]);
32          }
33          for (int[] p : permutations(numbers)) {
34              for (int i = 0; i < p.length; i++)
35                  System.out.print(p[i] + "␣");
36              System.out.println();
37          }
38      }
39  }
```

A.2.3 NewtonMethod

The `NewtonMethod` program implements approximation of $\sqrt{n}$ using Newton's method.

```
1   class NewtonMethod {
2       public static void main(String[] args) {
3           final double n = Double.parseDouble(args[0]);
4           final int maxSteps = Integer.parseInt(args[1]);
5
6           double q = 1;
7           for (int i = 0; i < maxSteps; i++)
8               q = (q + n / q) / 2;
9
10          System.out.println(q);
11      }
12  }
```