Masterarbeit

Porting the CACAO Virtual Machine to POWERPC64 and Coldfire

ausgeführt am

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

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

durch
Bakk. techn. Roland Richard Lezuo
Burggasse 35/1
1070 Wien

Oktober 2007

Abstract

CACAO is a freely available just in time compiler for the Java language. In the course of this master thesis code generators for the POWERPC64 and Coldfire architectures were developed.

This work describes is the generic structure of a CACAO code generators and elaborates implementation details. Benchmarking results of POWERPC64 compared with CACAO on x86_64 as well as SUN and IBM Java implementations will be presented.

Finally code simplification and future performance optimizations will be proposed by lessons learnt from this master thesis.

Kurzfassung

CACAO ist eine frei verfügbare virtuelle Maschine für Java welche auf einer just-in-time Compiler Architektur basiert. Im Zuge dieser Arbeit wurden Code Generatoren entwickelt die die Ausführung von Java Programmen auf POWERPC64 und Coldfire Prozessoren ermöglicht.

Es wird sowohl die generische Struktur eines CACAO Codegenerators beschrieben, als auch Details der Implementierungen die erstellt worden ist. Die Qualität der Arbeit wir aufgrund von Benchmarkergebnissen, in denen die POWERPC64 Architektur gegen IBM Java und CACAO auf x86_64, sowie SUN Java verglichen wird überprüft.

Schliesslich werden die aus der Arbeit erkannten Vereinfachungen der Codebasis sowie künftige Schritte zur Optimierung der Effizienz zusammengefasst.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 The Java language

1.1.1 The big picture

The Java programming language was designed as a language for micro controllers around 1990[12]. One design goal was easy portability to new micro controllers which lead to a virtual machine (VM) based implementation of the language. The Java language has a syntax very similar to C++ but its design is more influenced by smalltalk. In Java every object is derived from `java.lang.Object` so there is exactly one class hierarchy. Additionally so called primitive types are used which are not part of the class hierarchy and therefore disturb the orthogonally of the language. The rationale for primitive types is performance as no runtime type checks are needed when operating with them.

Today the Java language is widely used, primary for server side applications but also for cross platform solutions and web applets. It is also used for teaching computer science students, for example at Vienna University of Technology although doing so is not uncritizised [8]. On `http://freshmeat.net` 5267 out of 41.916 projects (as on 3.1.2007) are tagged as being implemented in the Java language. It has to be stated that these numbers are not representative as `freshmeat.net` mainly hosts UNIX software nonetheless it shows that Java has practical relevance.

The world famous "Hello world!" program in Java may look like listing 1.1. Although hello world programs tend to have little expressiveness describing features of a language they at least show the wordiness and Java is a very wordy language. This is not a bad thing as reading code tends to be easier when the language is more verbose, which is important for large scale projects with more than a handful of developers.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello␣world!");
  }
}
```

Listing 1.1: Java hello world

```
public class MemLeak {
  private static java.util.Vector buffer =
                                  new java.util.Vector();

  public static void main(String [] args) {
    for (;;) {
      java.lang.Integer i = new java.lang.Integer(1);
      buffer.add(i);
    }
  }
}
```

Listing 1.2: Java memory leak

1.1.2 Design goals

Java is a rather modern language and although compiled it has dynamic features as well. Java provides automatic garbage collection and does not provide pointer types. The rational behind this decisions was the believe that pointers are the main source of programming errors in C/C++. It is a common misbelief that Java eliminates memory leaks as a whole, it just eliminates some error classes where memory can leak. As counterexample for a Java program leaking memory see listing 1.2, which will be aborted by a `java.lang.OutOfMemoryError` exception. The problem with this program is the static Vector buffer which will always hold a valid reference to the Integer objects allocated in the loop. Therefore the garbage collector can not remove the objects and eventually all available memory will be used. In cases where static members are involved or objects with a very long lifespan the programmer has to take care that no resources are leaked.

A very interesting feature of the Java language is the security manager. The Java class library has hooks weaved in calling back `java.lang.SecurityManager` before executing code which may harm the system. The security manager of a program can be provided by the user so he has fine grained controls even for untrusted programs. That this sandbox can not be left easily is due to the virtual machine based implementation of the Java language which defines an interface to the host running the Java program. As any operations affecting the underlying system have to be passed through that interface it is possible to prevent possible harmful actions by blocking them at this defined border.

The Java platform is designed to be architecture neutral and portable. It therefore makes no assumption about host specific properties like machine word size or endianess. Instead a virtual machine interface is defined expected to be provided by the host. It is the responsibility of the virtual machine vendor to encapsulate hardware details in a way compatible with the expectations of a Java program executed. This includes word sizes, memory model and the semantics of built in functions or native interfaces. All programs written in the Java language can then be executed in a Java virtual machine (JVM) without any modification and are expected to behave identically.

It is stated that Java is high performance. This statement won't hold when comparing interpreted Java programs against native compiled binaries - an interpreter will always have some overhead [17, 11]. Nonetheless good performance of Java programs can be achieved by using just-in-time (JIT) compiler techniques, which may actually out perform native compiled code [23]. The majority of this paper will cover this topic. As a possibility to boost performance it is suggested to use the Java native interface (JNI)[1]. The JNI defines an interface to invoke native compiled code which may perform better than an implementation using Java. The main drawback of doing so is to loose the portability of the program.

Further the Java language supports threading at language level. This also lead to the class library being thread safe which eliminates a source of bugs with which for example C++ programmers are confronted when using standard template library (STL) containers in multi threaded applications.

Linking in Java happens dynamically. Classes are loaded and linked into the program when first accessed, not when the program is loaded. This is convenient when loading application via network as the initial start-up time is reduced.

1.2 Language features

Java offers so called primitive data types, types which are no objects. The integer types offered by the language are all signed, there is no unsigned data type available. The following integer types are offered:

[1]`http://java.sun.com/docs/books/jni/html/intro.html`

- `byte` (8 bit)
- `short` (16 bit)
- `int` (32 bit)
- `long` (64 bit)

The character type `char` is a 16 bit unsigned integer type as Java uses Unicode[2] as character encoding. Java also offers two floating point types, `float` (32 bit) and `double` (64 bit). Additionally a `boolean` type is offered which can be assigned the symbolic values `true` and `false` solely.

Objects are the main abstraction in Java programs. Objects can have members (primitive data types, other objects) and methods. Methods consist of statements and can operate on the members of an object. Java supports class inheritance with the limitation that a class can only inherit from one so called superclass. In addition so called interfaces are offered. An interface solely consists of method signatures and constants. A class implementing an interface has to provide an implementation for each interface method.

A more detailed introduction into the Java language is beyond the scope of this paper.

1.3 Executing Java programs

Java programs are compiled into so called class files by the Java compiler. The compilation process translates the textual representation of the program into an easier to parse intermediate form, called byte code. The byte code is not machine specific but operates on an abstract machine model. There are around 200 instructions, which a JVM executing the byte code has to implement. There are different approaches to implement a JVM. The earliest JVMs were interpreters, today just-in-time compilers are common. A just in time compiler creates machine code for the methods denoted as byte code and executes them directly on hardware. The name comes from the fact that the code is compiled directly before a method is executed. Most JVM implementations have an interpreter and compile only performance critical parts of the Java program.

[2] `http://www.unicode.org`

1.4 CACAO architecture

1.4.1 Overview

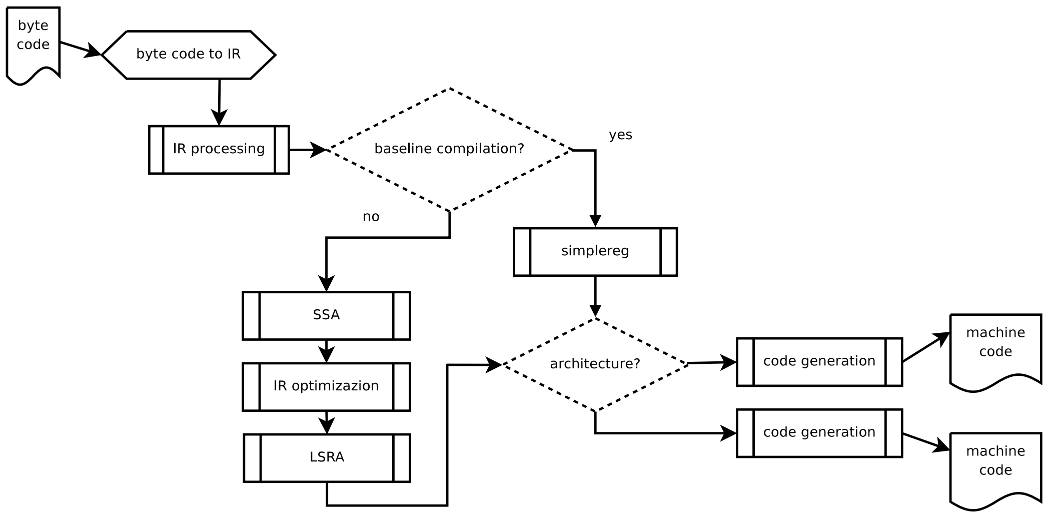


Figure 1.1: CACAO architecture overview

As described in [14], the CACAO system uses a compile only approach to execute bytecode. An interpreter has been added recently but a mixed mode where parts of the code are compiled and other parts are interpreted is not implemented. When initially loading a class no methods are compiled. This is done to save start-up time and due to the fact that many methods are never executed at all (when executing the DaCapo benchmark suite a total number of 86730 methods are loaded from class files but only 28240 are actually compiled. This is approximately 33%).

For each class loaded a so called `vtable` is created. A `vtable` contains pointers to the methods implemented by a class. Each object knows it's classes `vtable` and looks up the method entry point there on invocation. Interface methods need a second indirection to be looked up. Figure 1.3 shows the layout of a vtable. Method lookup tries to match the signature of the method to be called with a signature of a method referenced by the vtable. In Java not only the method name but also the types of the arguments define the signature. `java.lang.System.exit(I)V` is an example for a Java method signature. The letter `I` means that there is an integer argument `V` means a return value of type void. Figure 1.2 shows how to construct method signatures. Array types are denoted by prefixing the type with `[` (there are not arrays of type void however).

Type:	`int`	`long`	`float`	`double`	`byte`	`char`	`short`	`bool`	`void`
Specifier:	`I`	`J`	`F`	`D`	`B`	`C`	`S`	`Z`	`V`

Figure 1.2: Method signature specifiers

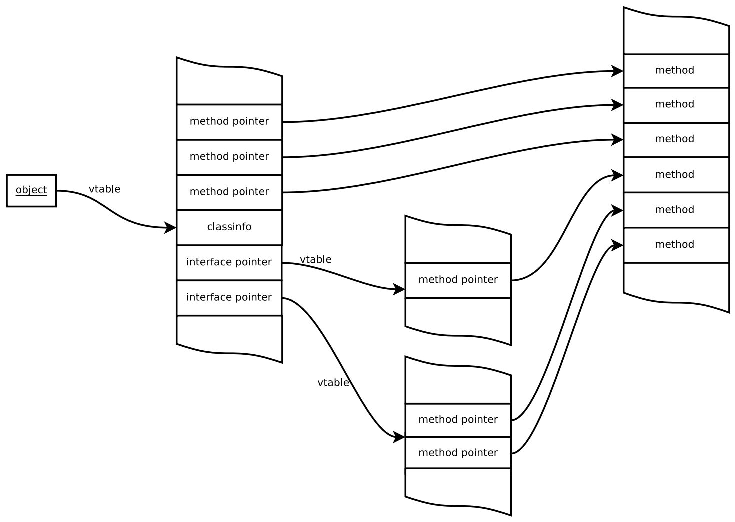


Figure 1.3: A CACAO vtable

Figure 1.1 gives an overview of the CACAO architecture. The byte code gets parsed and information is added. This results in an intermediate representation (IR) consisting of `ICMD`. For each method one of two possible compilers can be used. The baseline compiler is the simpler one and is used to quickly generate code. The optimizing compiler will only be used for performance critical methods. The compile time is increased, but much better code is generated. This pays back when the method is executed frequently[3]. It has to be noted that at the time of the writing only the baseline compiler is working. LSRA[24] and SSA[20] are currently implemented, recompilation framework is in place but still unused, this basically means that the current CACAO does not do any optimizations.

The `vtable` entries initially point to so called compiler stubs. The compiler stub when invoked determines the call site (from where it has been invoked), triggers compilation of the method which should have been called and updates the `vtable` accordingly (just-in-time compilation).

1.4.2 Startup and shutdown

The startup happens in `src/vm/vm.c` and initializes all modules needed by the runtime environment. The Java program is started with a single call to `vm_call_method`, which runs the main method of the given class. This method returns when the Java program main thread has terminated. When there are still active threads `vm_destroy` waits for all of them terminating before the whole Java program is terminated. In case of program termination caused by an uncaught exception a global variable called `exceptionptr` points to this exception. A stacktrace is generated and the VM terminates with an error exit code by calling `java.lang.System.exit(I)V`. Last mentioned function is called with a success exit code when no exception has occurred.

1.4.3 Linker

The linker is responsible for building the `vtable` of a class. It therefore has to load and link all interfaces a class implements as well as its superclass (recursively). When initializing the linker important classes are preloaded. The very first class to be loaded is `java.lang.Class`. Hardcoding classes to be loaded of course binds the VM somewhat to a class library layout and naming conventions. There are essential classes in a Java SE (Standard Edition) not found in a Java ME (Micro Edition). What follows is an incomplete list of classes the VM assumes to exist. They are used hardcoded from within the VM. Cacao supports GNU Classpath[3] (a free implementation of a Java class library), Sun CLDC[4] (a Java Micro Edition profile for mobile handsets), and recently support for Sun OpenJDK[5] class libraries - the open source version of the Java SE - has been added.

Some classes are only needed when using a specific class library implementation. Such classes are marked.

- `java.lang.Object`
- `java.lang.Class`
- `java.lang.ClassLoader`
- `java.lang.Cloneable`
- `java.lang.SecurityManager`
- `java.lang.String`
- `java.lang.System`

[3]`http://www.gnu.org/software/classpath/`
[4]`https://phoneme.dev.java.net/`
[5]`http://openjdk.java.net/`

- `java.lang.Thread`
- `java.lang.ThreadGroup`
- `java.lang.VMSystem`
- `java.lang.VMThread`
- `java.io.Serializable`
- `java.lang.Throwable`
- `java.lang.Error`
- `java.lang.LinkageError`
- `java.lang.NoClassDefFoundError`
- `java.lang.OutOfMemoryError`
- `java.lang.VirtualMachineError`
- `java.lang.AbstractMethodError (Java SE)`
- `java.lang.NoSuchMethodError (Java SE)`
- `java.lang.VMThrowable (GNU Classpath)`
- `java.lang.Exception`
- `java.lang.ClassCastException`
- `java.lang.ClassNotFoundException`
- `java.lang.IllegalArgumentException`
- `java.lang.IllegalMonitorStateException`
- `java.lang.Void (Java SE)`
- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Short`
- `java.lang.Integer`

- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.NullPointerException`
- `java.lang.StackTraceElement (Java SE)`
- `java.lang.reflect.Constructor (Java SE)`
- `java.lang.reflect.Field (Java SE)`
- `java.lang.reflect.Method (Java SE)`
- `java.security.PrivilegedAction (Java SE)`
- `java.util.Vector (Java SE)`

1.4.4 Method compilation

Whenever a not compiled method gets invoked the compiler stub ultimately invokes `jit_compile_intern` which controls the compilation (see figure 1.4). As a first step it is tested whether the method is native, in this case no code needs to be compiled, but a stub invoking the native function needs to be emitted. This stub code translates CACAO internal argument passing conventions to platform C ABI and calls the native function. When bytecode needs to be compiled the following phases are passed consecutively.

Parse

The basic blocks are determined. A basic block starts at an instruction which can be jumped to and ends at the first branching instruction or at the next basic block, whatever applies first. The stack usage of the method is simulated, and the Java byte code operands are translated into an internal representation. Complex instructions, like `dup_x2` are converted to a sequence of simpler instructions.

Stack analysis

For each basic block a symbolic evaluation of the code is performed, to check whether the stack underflows. At basic block boarders so called interface registers are used to merge control flow [13].

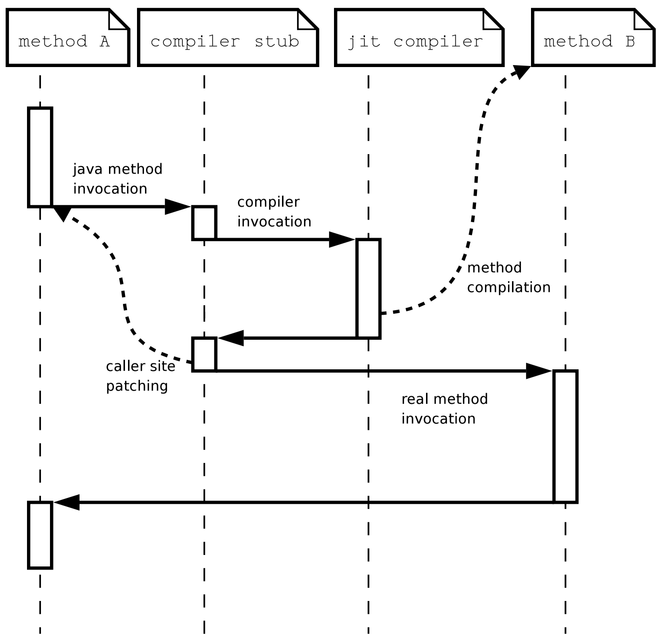


Figure 1.4: Method compilation

Control flow graph

Afterwards a control flow graph is built upon the basic blocks. The control flow graph is needed for the verifier. Code verification is mandatory for a Java virtual machine and specified in the JVM specification [6]. The verifier basically assures that all local variables references are valid, that parameters on the stack have correct types and no stack underflows occur.

Typecheck

As local variables are just index numbers in bytecode with no type information attached, they can be reused when their live ranges do not overlap. The verifier has to symbolically evaluate the code in more detail than needed for stack analysis.

Loop optimization

When enabled the compiler can try to optimize array bound checks in loops. The array bound check can be propagated outside the loop when the optimizer can prove that the index will stay in known borders and will therefore be only checked once [7].

[6]http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Static if-conversion

If-conversion is used to reduce the number of mispredicted branches, which lead to cache invalidation and therefore penalizes performance. To benefit from if-conversion a processor must support predicated opcodes. These are opcodes which are only executed when certain conditions (e.g. flags in the status registers) are met. For CACAO ARM and x86_64 are possible beneficiaries of if-conversion. Performance impact on real hardware is described in [4] for the Intel Itanium processor.

Inlining

Method inlining is another way to improve performance. Small methods are embedded in the caller instead of generating a method invocation. For Java this is especially true as a lot of getter methods are used which basically just return a value they read by a `getfield` instruction. Additionally a lot of initializers are called when creating an object. Most of them are empty or trivial. 80% of method invocations can be saved by inlining in some testcases[19].

Static, final and private methods are good candidates for inlining as the compiler knows which code will get called. Overwritten methods can not be inlined as the decision which implementation to call depends on the runtime type of the object. A potentially overwriteable method which is not overloaded on the other hand can be inlined as well. Care has to be taken when new classes get loaded by the program because some class may overload the method in which case the inlining has to be undone - by recompiling the method. All speculative inlinings need to be kept track of to handle this case.

Block reordering

All basic blocks get reordered with respect to their calling frequency. This is done to increase cache locality and described in [9].

Register allocation

Every slot from the stack machine model gets assigned a register on the real hardware. As there is only a limited number of registers available and the number of used stack slots may exceed the registers of a processor register allocation is crucial for performance. The register allocator in CACAO is replaceable, but for now only a very simple allocator is implemented.

The simple register allocator first allocates interface registers[13] and does register precoloring for argument and return registers. Next so called temporaries and finally local variables are allocated. When there are not enough registers available the stack slot is allocated in memory and has to be swapped in before each usage and stored back afterwards. More sophisticated register allocators may avoid some spilling and copying but take more time and therefore hurt performance more than they benefit as compile time is part of method run time (at least for the first run). [13].

LSRA Linear Scan Register Allocation is a fast register allocator suitable for just in time compilation described in [15]. It's results are comparable with more expensive graph coloring based allocators. CACAO will only use LSRA when compiling a method using the optimizing compiler, the baseline compiler uses the simple register allocator described earlier.

Code generation

The last step of compilation is the code generation. CACAO supports multiple platforms and all phases so far have been generic. Code generation basically iterates all bytecodes of a method and writes machine code implementing theses bytecodes. Chapter 3 will examine the work of the code generator in more detail.

1.5 Java byte code

The Java compiler does not produce machine code which can be executed on the host CPU directly but an intermediate form called byte code. There are around 200 instructions defined which a virtual machine has to execute according to their specification. Instructions are specified as operations on an operand stack. This type of virtual machine is therefore called a stack machine. The program listing 1.3 manipulates the stack as shown in figure 1.5. The opcode `iconst_3` pushes the integer 3 on top of the stack, `iconst_5` pushes 5, `iadd` takes the two topmost elements of the stack, adds them and pushes the result back. The stack growing from bottom to top and its initial state is unknown but also unimportant.

The JVM internally uses 32 bit for each stack slot. Therefore the primitive types `boolean, char, byte, short, int` and `float` take exactly one stack slot, `long` and `double` take two. Each instruction consists of at least one byte (the opcode itself) followed by optional arguments. The `getfield` instruction for example uses two argument bytes indexing the field in the object, while

```
        iconst_3
        iconst_5
        iadd
```

Listing 1.3: Stack operations

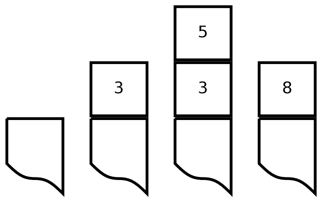


Figure 1.5: Stack changes

it's object reference is passed on the stack. The arithmetic instructions are additionally tagged with the type of the arguments (e.g. `iadd` adds two `int` whereas `ladd` adds two `long` values). Figure 1.6 shows the type prefixes.

The following sections list the most important opcodes defined by the virtual machine definition grouped by usage.

Arithmetic opcodes

- `iadd, ladd, fadd, dadd` - addition
- `isub, lsub, fsub, dsub` - subtraction
- `imul, lmul, fmul, dmul` - multiplication
- `idiv, ldiv, fdiv, ddiv` - division
- `irem, lrem, frem, drem` - remainder of division
- `ineg, lneg, fneg, dneg` - negation

Bit instructions

- `ishl, lshl` - shift left

Prefix:	i	l	f	d	b	c	s	a
Type:	int	long	float	double	byte	char	short	address

Figure 1.6: Instruction type prefixes

- `ishr, lshr` - shift right
- `iushr, lushr` - shift right unsigned (logical shift)
- `ior, lor` - logical or
- `iand, land` - logical and
- `ixor, lxor` - logical exclusive or

Comparisons

- `dcmpg, dcmpl` - compare doubles
- `fcmpg, fcmpl` - compare floats
- `lcmp` - compare longs

Conversions

- `i2l, i2f, i2d, l2f, l2d, f2d` - widening conversions
- `i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f` - narrowing conversions

Flow control

- `ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonull` - takes one `int` value from stack, compare with 0, and jump at a given offset if true.
- `if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge` - take two `int` values from stack, compare and jump at given offset.
- `if_acmpeq, if_acmpne` - take two addresses from stack, compare and jump at given offset.
- `lookupswitch` - used to implement the switch/case statement of the Java language.
- `tableswitch` - similar to lookupswitch
- `goto, goto_w` - jump at given address

- `jsr, jsr_w` - jump to subroutine, store return address on stack
- `ret` - return from subroutine

Load and store

The virtual machine additionally knows the concept of local variables. These are numbered memory locations, not located on the stack. Local variables are generated by the compiler for each method and can be thought of as an fixed size array. Local variables are used to pass method arguments and hold temporary variables needed during calculations. The following operations are used to access them.

- `iload, lload, fload, dload, aload` - load value from given variable and stores on stack.
- `bipush, sipush` - push a constant byte/short onto the stack.
- `ldc, ldc2_w` - push a 32/64 bit value onto the stack.
- `istore, lstore, fstore, dstore, astore` - take value off stack and stores it in local variable.
- `wide` - most opcodes use an 8 bit offset to number local variables. If such an opcode is preceded by wide it uses a 16 bit offset.

Some instructions used for optimization - like the already mentioned `iconst_3`, which stores an integer value of 3 onto the stack - have been skipped for clarity.

Stack operations

- `pop, pop2` - remove the (two) topmost element(s) from the stack.
- `dup, dup2` - duplicate the topmost element (twice).
- `dup_x1, dup2_x1, dup_x2, dup2_x2` - duplicate one or two words and insert them one or two words below the top of stack.
- `swap` - swap the two topmost elements.

Object operations

- `new` - create an instance of a class
- `instanceof` - test whether an object is of class type
- `checkcast` - similar to instanceof

Field operations

- `getfield, getstatic` - load on stack content of (static) member
- `putfield, putstatic` - store value in (static) member

Method invocation

- `invokevirtual` - invoke a method using vtable lookup.
- `invokeinterface` - invoke a method looking up in interface table.
- `invokespecial` - used to invoke initialization methods and methods from superclass and private methods.
- `invokestatic` - invoke a static method
- `ireturn, lreturn, freturn, dreturn, areturn` - return value of given type.
- `return` - return void type.

Array operations

- `newarray, anewarray, mulitnewarray` - create a new array.
- `baload, caload, saload, iaload, laload, faload, daload, aaload` - load from array.
- `bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore` - store into array.
- `arraylength` - store length of array onto stack.

Miscellaneous operations

- `athrow` - throw an exception.
- `monitorenter, monitorexit` - each object has a monitor associated with it, when entered it blocks access from concurrent threads until left.

A detailed definition of the Java instruction set has been published by SUN (The Java^{TM}Virtual Machine Specification, Second Edition) which is also available online at `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`.

Chapter 2

Related work

This chapter will introduce a few important other Java VMs. Sun's HotSpot, IBM's J9 and the open source Jikes RVM. The architecture of their code generator will be compared to CACAO's design. Further interesting VMs will be introduced. Kaffee an open source Java VM, Parrot the Perl 6 implementation and Mono a C# VM.

2.1 SUN HotSpot

SUN's own Java VM implementation features two different implementations[1]. Client and server mode. Client mode JIT compiler focuses on local optimizations, whereas server mode takes the global situation into account at much higher compile time costs. Code generators are available for SPARC, x86 and AMD64. An rough overview of the architecture is given in figure 2.1.

CACAO does not feature two different operation modes (client, server),

[1]`http://java.sun.com/products/hotspot/whitepaper.html#client`

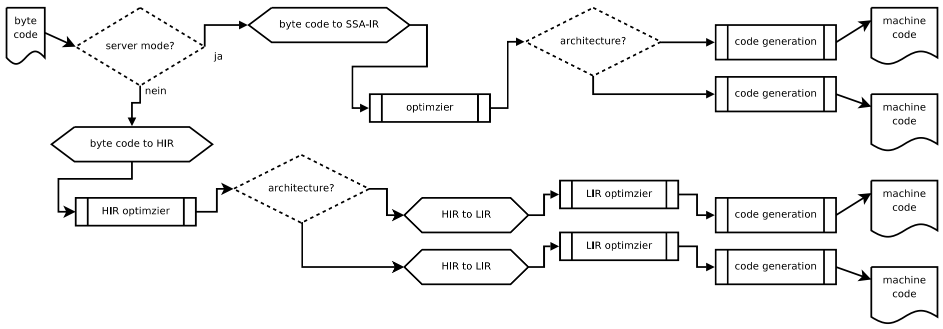


Figure 2.1: SUN HotSpot code generation

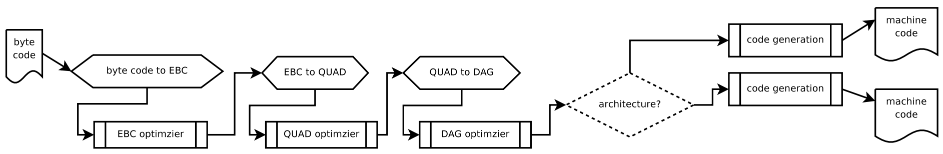


Figure 2.2: IBM J9 code generation

the compiler has only one IR, whereas HotSpot has two IRs (in client mode). In client mode HotSpot differs hardware architecture earlier than CACAO and has a platform specific LIR.

2.2 IBM J9

The IBM J9 virtual machine is derived from the Sun Classic VM, but has been further developed[10]. Code generators are available for PPC, x86 and AMD64. The JIT compiler transforms the byte code into extended byte code (EBC) which is still stack based. After optimizing the EBC (e.g. inlining) this form is transformed into a register based quadruple (QUAD) representation. Common subexpression elimination is among other things performed on the finer grained quadruple code which is the transformed into a directed acyclic graph (DAG) which gets scheduled and is then transformed back into quadruple code for code generation. This architecture of the optimizing compiler is shown in figure 2.2.

IBM's J9 even features three IRs for the optimizing compiler. Like in CACAO most stages are generic, only the codegenerator is architecture specific.

2.3 Jikes RVM

The Jikes RVM[2] was initially started as project Jalapeño[1] by IBM in 1997. It has been put under an open source license in 2001 and is since then actively maintained. The main focus of the project is to provide a research VM. New ideas shall be quickly implementable on top of Jikes RVM. Nonetheless Jikes RVM is a fast VM too. Most parts of the VM are implemented in the Java language as well, only around 1000 lines of C code are needed for bootstrapping. Jikes RVM features an unoptimizing baseline compiler as well as an highly sophisticated optimizing compiler. Code generators are available for PPC and x86.

[2]`http://jikesrvm.org`

```
protected final void emit_iadd() {
  popInt(T0);
  popInt(T1);
  asm.emitADD(T2, T1, T0);
  pushInt(T2);
}
```

Listing 2.1: Jikes RVM baseline add instruction

```
public final void emitADD(int RT, int RA, int RB) {
  int mi = ADDtemplate | RT << 21 | RA << 16 | RB << 11;
  mIP++;
  mc.addInstruction(mi);
}
```

Listing 2.2: Jikes RVM baseline assembler add instruction

Listing 2.1 shows a baseline compiler code snippet taken from `/rvm/src/org/jikesrvm/compilers/baseline/ppc/VM_Compiler.java` emitting the code to generate a Java `iadd` instruction. The code snippet generating the assembler code is shown in listing 2.2 and is taken from `src/org/jikesrvm/compilers/common/assembler/ppc/VM_Assembler.java`

The optimizing compiler is using a different assembler, generated by a BURS[16] implementation. Listing ref 2.3 shows the definition of an add instruction take from `rvm/src-generated/opt-burs/ppc/PPC_Alu32.rules`

An overview of the code generator is given in figure2.3. Jikes's baseline compiler is similar to CACAO, its optimizing compiler features three IRs with one being hardware specific.

```
r: LONG_ADD(r,r)
20
EMIT_INSTRUCTION
LONG_ADD( P(p), Binary.getResult(P(p)),
          R(Binary.getVal1(P(p))),
          R(Binary.getVal2(P(p))) );
```

Listing 2.3: Jikes RVM optimzing add instruction

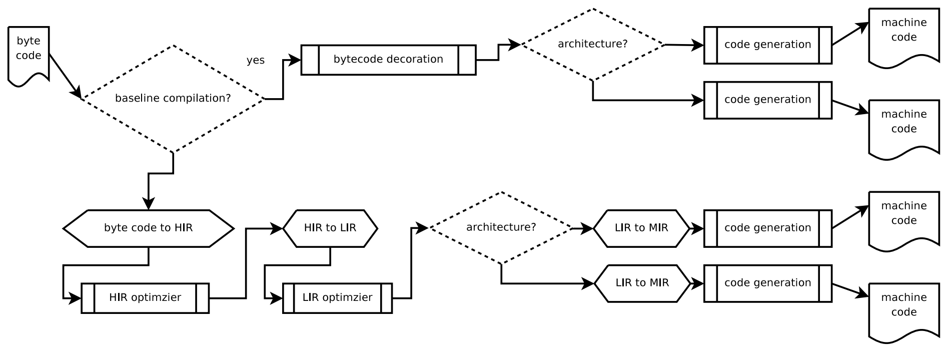


Figure 2.3: Jikes RVM code generation

2.4 Kaffe

The Kaffe[3] project is one of the oldest free (GPL licensed) Java implementations around. Version 0.6 was released in 1996, version 1.1.7 was released in march 2007. Kaffe features an interpreter and a JIT for some architectures. It also supports different flavour of Unixes including Linux and FreeBSD. Version 1.1.7 includes JIT for ALPHA, ARM, X86, X86_64, PPC, M68K MIPS, S390 and others. Kaffe has 2 different JIT engines, in this paper only JIT3 (the newer one) is covered. An architectural overview is shown in 2.4.

As a first step of code generation Kaffe translates the Java byte code into an Kaffe specific intermediate format. This transformation is driven by `kaffe/kaffevm/kaffe.def`. An example of such a transformation rule is given in listing 2.4, which shows the transformation for the Java bytecode operation `IADD`.

The translation of this sequence of opcodes into machine code is implemented in `kaffe/kaffevm/jit3/icode.c`. For each opcode processed a callback into an architecture specific code generator is invoked. This translation process knows about machine details and only invokes callbacks implemented by the code generator. This is controlled by a bunch of `HAVE_` macros, e.g. `HAVE_add_int`. An example of such a callback invocation is given in listing 2.5.

The Code generation is driven by a machine definition file in the config sub directory of the Kaffe source. For m68k the file is located at `config/m68k/jit3-m68k.def`. As example the generation of machine code performing integer addition is given in listing 2.6.

The main difference to the CACAO architecture is the highly configurable but generic code generator which drives machine code output for each

[3]`http://kaffe.org`

```
define_insn(IADD)
{
  /*
   * ..., val1, val2 -> ..., val1+val2
   */
  trace_jcode ("iadd\n");

  check_stack_int(0);
  check_stack_int(1);

  add_int(stack(1), rstack(1), rstack(0));
  pop(1);
}
```

Listing 2.4: Kaffe intermediate example

```
void
add_int(SlotInfo* dst, SlotInfo* src, SlotInfo* src2)
{
#if defined(HAVE_add_int_const)
  if (slot_type(src) == Tconst) {
    add_int_const(dst, src2, slot_value(src)->i);
  }
  else if (slot_type(src2) == Tconst) {
    add_int_const(dst, src, slot_value(src2)->i);
  }
  else
#endif
  _add_int(dst, src, src2);
}
```

Listing 2.5: Kaffe intermediate transformation

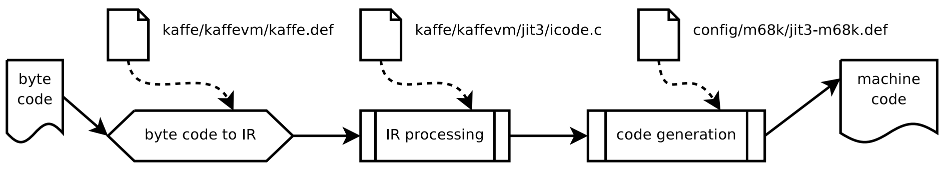


Figure 2.4: Kaffe code generation

```
define_insn(add_int, addi_RRR)
{
  int r = rreg_int(2);
  int w = rwreg_int(0);

  assert(rreg_int(1) == w);

  op_addl_dd(r, w);
}

static inline void
op_addl_dd(int src, int dst)
{
  debug(("addl␣%s,␣%s\n", regname(src), regname(dst)));
  assert_dreg(src);
  assert_dreg(dst);
  WOUT(0xD080 | (dst << 9) | (MODE_d << 3) | (src & 7));
}
```

Listing 2.6: Kaffe codegenerator definition example

platform. The code generator provides a bunch of functions emitting the assembler instructions, but contains no loop iterating the byte code.

2.5 Parrot

Parrot [4] 0.0.1 was released in September 2001, the last release as of now is 0.4.12 which has been released in May 2007. Parrot was originally started as implementation of Perl 6 but now focuses on providing a virtual machine for many dynamically typed languages as Python and Lua. Parrot is a register based virtual machine featuring a lot of high level instructions, including e.g. an implementation for the coversine function[5] or string concatenation. Parrot also tries to make objects implemented in different languages interoperable. The JIT compiler is ported for alpha, arm, x86, IA64, PC, sun4, MIPS and HPPA.

The Parrot JIT compiler is written in a markup language processed by a Perl script to generate C source code. An example snippet from `src/jit/ppc/jit_emit.h` is given in listing 2.7 and shows how an add in-

[4] `http://www.parrotcode.org`

[5] $1 - sin(x)$

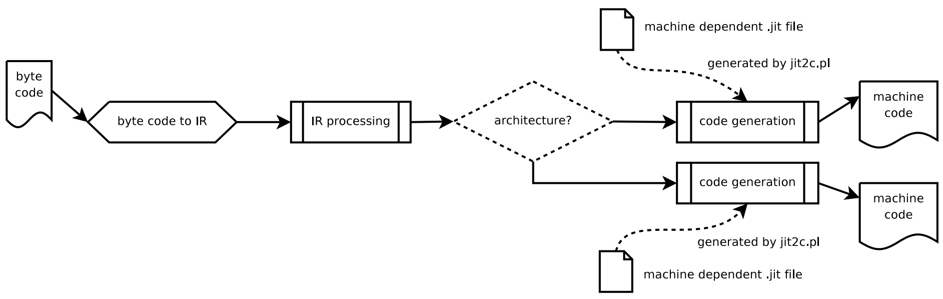


Figure 2.5: Parrot code generation

```
#define jit_emit_3reg_x(pc, opcode, D, A, B, type, Rc) \
  *(pc++) = opcode << 2 | D >> 3; \
  *(pc++) = (char)(D << 5 | A); \
  *(pc++) = B << 3 | type >> 7; \
  *(pc++) = (char)(type << 1 | Rc);

#define jit_emit_add_rrr(pc, D, A, B) \
  jit_emit_3reg(pc, 31, D, A, B, 0, 266, 0);
```

Listing 2.7: Parrot codegeneration example

struction is emitted. Code generation is driven by `src/jit/ppc/core.jit` which implements Parrots intermediate instructions, an example is given in listing 2.8 and shows generation of an addition, the `binop_x_x` template expands to `jit_emit_add_rrr`, this expansion is done by the Perl script `tools/build/jit2c.pl`. An overview is shown in figure 2.5.

The main difference to CACAO is the introduction of another intermediate language to describe the code generators. In addition Parrot is a register based VM where CACAO is stack based and Parrot is tailored towards dynamically typed languages.

```
Parrot_add_i_i {
  binop_x_x s/<_N>/_i/ s/<op>/add/ s/<s1>/ISR1/
            s/<s2>/ISR2/ s/<T>/INT/
}
```

Listing 2.8: Parrot JIT definition

```
lreg: OP_I8CONST {
 MONO_EMIT_NEW_ICONST(s,state->reg1,tree->inst_ls_word);
 MONO_EMIT_NEW_ICONST(s,state->reg2,tree->inst_ms_word);
}
```

Listing 2.9: Mini generic instruction selection

2.6 Mono

The Mono[6] project was initiated by Miguel de Icaza in 2001. Novell[7] picked up the idea and Mono 1.0 was released in June 2004. The Mono project aims to provide a C# compiler and a .NET runtime under a free software license. The Mono system is available for different operation systems and features a JIT compiler for all supported platforms. As of today s390, SPARC32, PPC, x86, x86_64, IA64, arm, alpha and MIPS are supported. Mono code generation is driven by the function `mini_method_compile` which first of all reads in the .NET IL (intermediate language) and constructs a mono specific intermediate representation (IR).

This IR is then transformed by `mini_select_instructions` into a machine dependent optimized IR. An iburg like instruction selector[6] called mini has been implemented for this purpose. There are also generic rules, e.g. to map 64 bit operations to 32 bit operations, or to use floating point software emulation when appropriate. Listing 2.9 shows an example rule for loading a 64 bit constant on a 32 bit architecture (`mono/mini/inssel-long32.brg`).

`mono_codegen` finally produces machine instruction from the IR by calling an architecture specific backend (e.g `mono/mono/mini/mini-ppc.c`). The code generator iterates all IR opcodes of a basic block and selects machine instructions by using a big switch statement. This process is visualized in figure 2.6.

The machine instructions are described in a CPU specific file, and example taken from `mono/mono/mini/cpu-g4.md` may look like `add: dest:i src1:i src2:i len:4`, which describes the `add` instruction and its register usage. This information is used by the Mono registers allocator and instruction scheduler, but is otherwise unrelated to instruction selection.

The biggest difference between CACAO and Mono is the machine dependent transformation on the IR performed by Mono. This reduces the size of each code generator as e.g. 64 bit arithmetic byte code instruction don't need to be implemented by each 32 bit generator. In CACAO each 32 bit

[6]http://www.mono-project.com
[7]http://www.novell.com

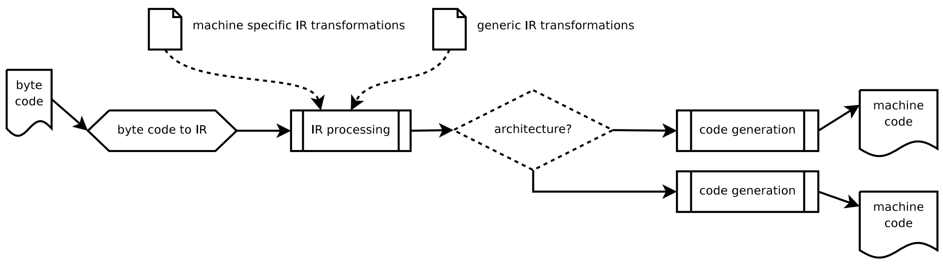


Figure 2.6: Mono code generation

code generator implements 64 bit arithmetic.

2.7 Summary

Despite their similar function code generators differ a lot. The number of introduced IRs is related to optimizations implemented. Only the commercial VMs implement the complex optimizations demanding different IRs. Some VM vendors decided to implement an instruction selector, while others iterate the IR and generate machine code by a switch statement. Another design decision is when to consider hardware specific aspects. Most code generators try to stay generic as long as possible. CACAO does not use an instruction selector. Therefore the code generators source size is huge, but mostly trivial. By directly implementing the machine code generating in C no additional language (like Parrot's JIT definition) has to be learned for development.

Chapter 3

Code generators for CACAO

CACAO provides code generators for many platforms. Currently code generators for ALPHA (FreeBSD, Linux), ARM (Linux) i386 (Cygwin, Darwin, FreeBSD Linux), MIPS (Irix, Linux), POWERPC (Darwin, Linux, NetBSD), SPARC64 (Linux), x86_64 (Linux) and s390 (Linux) are available. When building CACAO for a platform the according code generator is linked statically. To be able to easily exchange code generators an internal interface is defined. A code generator has to implement this interface.

3.1 Symbols needed by CACAO

The following symbols need to be defined to link a code generator with CACAO. As CACAO is under development this list is not stable, probably some symbols have been renamed and/or added, removed. The functions prefixed with `md_` are machine dependent, but also often operating system depended, they have been factored out to make porting from one operating system to another easier. The `asm_` prefixed functions are implemented in assembler language for various reasons. The signatures of the functions do not match reality, none of them is void actually, the notation has been chosen to differ between functions and global variables, the signature is skipped for clarity.

- `void codegen(void) {}`
  Main driver function for the code generator.

- `void createnativestub(void) {}`
  Generates code which translates CACAO internal argument format to platform C ABI format and calls a native function.

- `void createcompilerstub(void) {}`
  Generates code which invokes the compiler when a not compiled method

is invoked.

- `void md_init(void) {}`
  Initializations needed for the machine dependent part.

- `void md_get_method_patch_address(void) {}`
  When a method has been compiled the method entry point has to be patched so that further invocations execute the compiled code directly. This function returns a pointer to the memory location of the method entry point and differs types of method invocations.

- `void md_icacheflush(void) {}`
  Invalidates the processors instruction cache. This is needed when code patching took place as the processor would execute obsolete instructions else.

- `void md_stacktrace_get_returnaddress(void) {}`
  Returns the return address of the current stackframe. The caller does not need to know about the details of the platform. This is needed for the stacktrace generating code.

- `void md_codegen_get_pv_from_pc(void) {}`
  Each CACAO method has a procedure vector, this is its start address (entry point). This function takes a program counter as argument and returns the start address of the method.

- `void md_signal_handler_sigsegv(void) {}`
  Cacao uses illegal memory access to implement exceptions. The signal handler gets called by the operating system in such a case and activates exception handling.

- `void md_codegen_patch_branch(void) {}`
  This function changes the displacement of a branching instruction. The caller does not need to know details about the platform.

- `void md_cacheflush(void) {}`
  Invalidates the CPU data cache, this is needed when data patching took place otherwise the processor would work with obsolete values.

- `void md_param_alloc(void) {}`
  This function preallocates stack slots to argument registers for method invocations. This prevents copying values to argument registers by holding the arguments in the correct registers in the first place.

- `void md_return_alloc(void) {}`
  Preallocates the register holding the return value of a native function call.

- `int nregdescint;`
  An array containing the usage description for each integer register. A register description is one of: reserved, argument, saved and temporary. The register allocator needs this information. Saved registers are callee saved, whereas the others are caller saved.

- `int nregdescfloat;`
  Same as `nregdescint` for floating point registers.

- `void compare_and_swap(void) {}`
  Compare and swap (CAS) is a universal locking primitive [22] and can be used to implement lot of non-blocking algorithm. In CACAO it is used to implement method and object locks. Not all platforms are capable to provide an atomic implementation as it would be needed.

- `#define STORE_ORDER_BARRIER()`
  Guarantees that store operation preceding the barrier are finished before stores following the barrier.

- `#define MEMORY_BARRIER_BEFORE_ATOMIC()`
  Needed to implement the Java memory model as described in chapter 17 of the Java Language Specification [1].

- `#define MEMORY_BARRIER_AFTER_ATOMIC()`
  See `MEMORY_BARRIER_BEFORE_ATOMIC()`.

- `#define MEMORY_BARRIER()`
  See `MEMORY_BARRIER_BEFORE_ATOMIC()`.

- `#define PATCHER_CALL_SIZE`
  The size of a patcher in bytes. Patcher are described in 3.6.

- `#define BRANCH_NOPS`
  Generates enough `nop` instructions to overwrite it with a branch instruction later.

- `void asm_vm_call_method{}`
  Called when a Java method is invoked by C code. Mainly performs argument adaption, from C ABI to CACAO intern argument handling.

[1] `http://java.sun.com/docs/books/jls/third_edition/html/memory.html`

- `void asm_vm_call_method_[int|long|float|double] {}`
  These four functions are specialized versions of the above. The type in the name indicates the type of the return value expected.

- `void asm_vm_call_method_exception_handler {}`
  Needed to propagate exceptions beyond an `asm_vm_call_method` stack-frame. See 3.7 for more details.

- `void asm_call_jit_compiler {}`
  Invokes the JIT compiler from within JIT code and takes care of the calling conventions.

- `void asm_criticalsections {}`
  A data structure needed by `asm_getclassvalues_atomic`, contains start and end address of critical sections.

- `void asm_getclassvalues_atomic {}`
  Subtype testing in CACAO is implemented by relative numbering [18] which uses the post order numbering of an inheritance tree. These values need to be recomputed when new classes get loaded and may not be read while class loading is performed. This function reads a classes values and sets up a critical section around itself.

- `void asm_abstractmethoderror {}`
  This stub gets called when an abstract method is executed (which is prohibited in Java) and throws an `AbstractMethodError` exception.

3.2 Register allocation

When the code generator is invoked, all stack slots used by the method have been assigned a register or a memory location. The code generator has to emit code running on register based hardware, so each operand has to be loaded into a register before usage. For that reason three registers (of each type) are reserved for the code generator. Every bytecode instruction can be implemented using a maximum of three registers. The reserved registers are called `REG_ITMP1`, `REG_ITMP2` and `REG_ITMP3` for integer operations and `REG_FTMP1`, `REG_FTMP2`, `REG_FTMP3` for floating point operations. The `emit_load_s1`, `emit_load_s2` and `emit_load_s3` functions are used to load operands for the current bytecode operation. These functions take a temporary register as argument and return the register holding the operand. When the operand was in memory it has been loaded into the given temporary registers, otherwise the register assigned for the operand by the register allocator

```
case ICMD_IADD:
  s1 = emit_load_s1(jd, iptr, REG_ITMP1);
  s2 = emit_load_s2(jd, iptr, REG_ITMP2);
  d = codegen_reg_of_dst(jd, iptr, REG_ITMP2);
  M_IADD(s1, s2, d);
  M_EXTSW(d,d);
  emit_store_dst(jd, iptr, d);
  break;
```

Listing 3.1: Codegeneration for `iadd`

is returned. An `emit_store` function is used to finally store the result in either destination register or memory location. The destination register of an operation is returned by the function `codegen_reg_of_dst`. See listing 3.1 for an example showing the implementation of the `iadd` opcode on PPC64.

By using this API the code generator is completely decoupled from register allocation, with the costs of statically assigning three registers dedicated to the code generator.

3.3 Code generation macros

The code generator iterates over all bytecode instructions of the method to be compiled selecting machine code by a switch statement. The generated machine code is written to temporary memory and afterwards copied to an executable memory location. The assembler instructions are generated by macros, so care has to be taken with side effects of arguments which could be evaluated twice. To achieve a better conformity of the code generators all platforms try to use similar macro names originally inspired by alpha naming conventions. Listing 3.1 shows the implementation of the `iadd` operation for POWERPC64. One can see the load of the two operands. The macros `M_IADD` emits the machine code for an addition, `M_EXTSW` is needed for sign extension and is platform specific. Finally the result is stored in the according register. `jd` and `iptr` contain information about the state of the JIT compiler and the current instruction processed. The implementation of the `M_IADD` is shown in listing 3.2.

```
#define M_OP3(opcode,y,oe,rc,d,a,b) \
  do { \
    *((u4 *)cd->mcodeptr) = (((opcode)<<26) | ((d)<<21)\
        | ((a)<<16) | ((b)<<11) | ((oe)<<10) | ((y)<<1)\
        | (rc)); \
    cd->mcodeptr += 4; \
  } while (0)

#define M_IADD(a,b,c) M_LADD(a,b,c)
#define M_LADD(a,b,c) M_OP3(31, 266, 0, 0, c, a, b)
```

Listing 3.2: Codegeneration macros

3.4 Post compile time code patching

One reason the code generated is written into a buffer is due to unresolved jumps. Imagine a forward jump in a method. The target address points into code still not generated and the compiler does not know the exact offset in advance as it depends on the opcodes in between. Additionally the offset also depends on the whereabouts (register, memory) of the operands used by the instructions in between as this may change their size. For that reason a post pass has been added to code generation which patches the code after generation. A function named `codegen_add_branch_ref` is responsible for collecting branch addresses which may be unresolved at compile time. As an argument it takes the target basic block. When the jump can not be resolved the branch displacement will be patched in the post compilation phase using the function `md_codegen_patch_branch`. By using the machine dependent patching function the post compilation phase is platform independent.

3.5 Data segment

The generated code needs lots of constant values, some embedded in the program executed but also function entry addresses. Some architectures can load immediate values large enough to hold all constant values needed, other architectures have immediates not covering the whole range. For that reason a data segment holding constant values is generated for each method. The data segment is built together with code and later both are copied into memory consecutively. The first operation of the method is reference by the procedure vector (`pv`) in CACAO. Further instructions are at memory locations above the `pv` the data segment is below the `pv` (see figure 3.1).

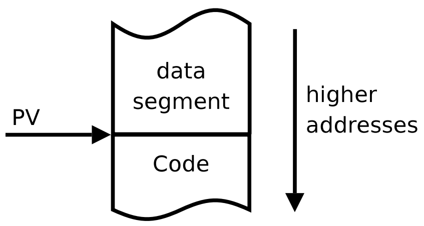


Figure 3.1: Data segment layout

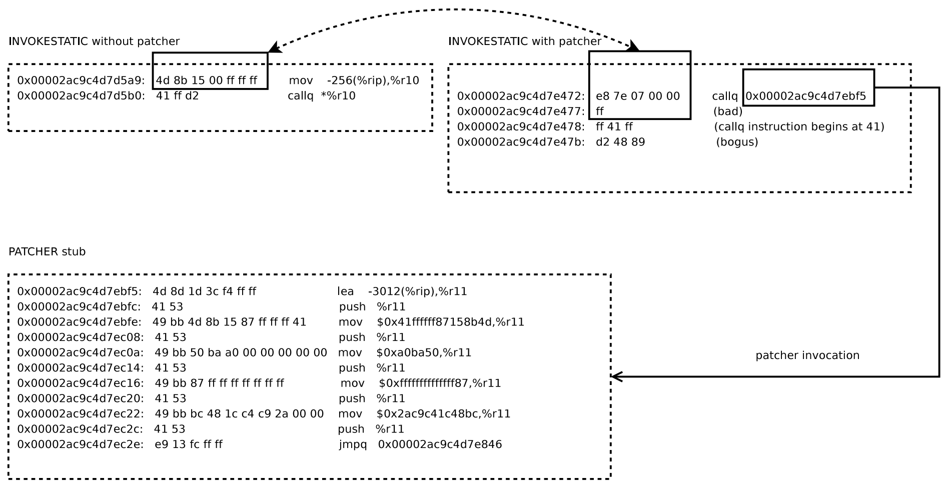
When a constant exceeding the size of immediate operands is needed it is loaded from the data segment by using an indirect load operation. In addition a method header is stored in the data segment which points to a C data structure describing the method (e.g. stack size, argument types, signature). The exception table and the line number table are stored in the data segment as well.

3.6 Runtime code patching - Patchers

Some references can not be resolved at compile time at all. An example would be a `getstatic` instruction, which loads a static field from a given class. The class may be unresolved when code for `getstatic` was generated in which case the runtime system has to load and initialize the class, resolve the address of the member and return that value. For this purpose so called patchers are embedded into the generated code which remove themselves after execution as they are only needed once.

The old approach

Old versions of CACAO implement patchers as branch instructions and jump to a generated patcher stub setting up a special stack frame and invoke `asm_patcher_wrapper` thereafter. `asm_patcher_wrapper` takes care of register storing and restoring and calls `patcher_wrapper` which is written in C. Figure 3.2 shows the assembler code of an embedded patcher. These snippets are from x86_64 and show an `invokestatic` with and without patcher. The leftmost snippet shows an invocation with known method entry point. The right snippet shows the code generated when the entry point was unknown at compile time. A branch to a patcher has been emitted. The patcher invocation is one byte shorter than the original invocation, therefore the assembler output is disturbed by the last byte of the original machine code (`0xff`). One can see the sequence `0x41 0xff 0xd2` which is the `callq` of the original



```
0x00002ac9c4d7d5a9:  4d 8b 15 00 ff ff ff      mov    -256(%rip),%r10
0x00002ac9c4d7d5b0:  41 ff d2                  callq  *%r10
```

```
0x00002ac9c4d7e472:  e8 7e 07 00 00    callq 0x00002ac9c4d7ebf5
0x00002ac9c4d7e477:  ff                (bad)
0x00002ac9c4d7e478:  ff 41 ff          (callq instruction begins at 41)
0x00002ac9c4d7e47b:  d2 48 89          (bogus)
```

```
0x00002ac9c4d7ebf5:  4d 8d 1d 3c f4 ff ff             lea    -3012(%rip),%r11
0x00002ac9c4d7ebfc:  41 53                            push   %r11
0x00002ac9c4d7ebfe:  49 bb 4d 8b 15 87 ff ff ff 41    mov    $0x41ffffff87158b4d,%r11
0x00002ac9c4d7ec08:  41 53                            push   %r11
0x00002ac9c4d7ec0a:  49 bb 50 ba a0 00 00 00 00 00    mov    $0xa0ba50,%r11
0x00002ac9c4d7ec14:  41 53                            push   %r11
0x00002ac9c4d7ec16:  49 bb 87 ff ff ff ff ff ff ff    mov    $0xffffffffffffff87,%r11
0x00002ac9c4d7ec20:  41 53                            push   %r11
0x00002ac9c4d7ec22:  49 bb bc 48 1c c4 c9 2a 00 00    mov    $0x2ac9c41c48bc,%r11
0x00002ac9c4d7ec2c:  41 53                            push   %r11
0x00002ac9c4d7ec2e:  e9 13 fc ff ff                   jmpq   0x00002ac9c4d7e846
```

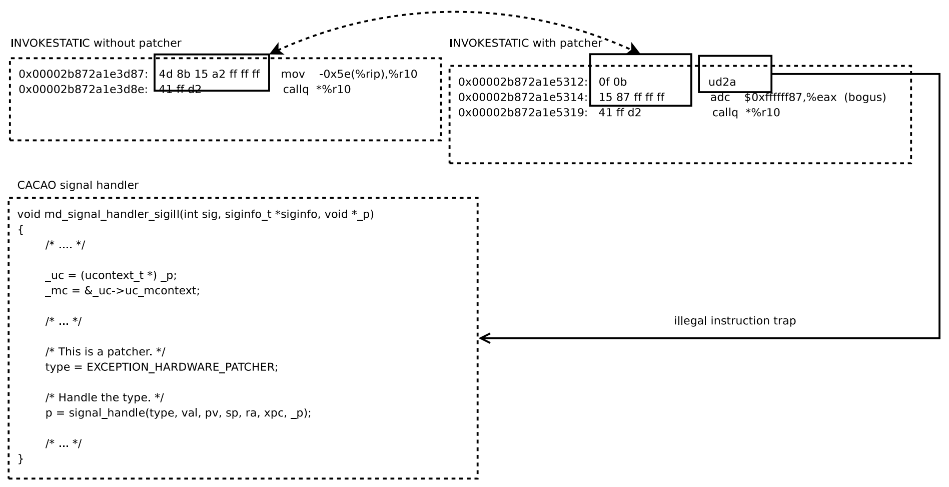
Figure 3.2: Patcher assembler output (old)

code. The patcher jumps to the shown patcher stub. This stub sets up the special stack layout used by patchers and invokes the patcher wrapper. This wrapper finally calls the patcher needed. The patcher is implemented in C code and afterwards the patcher invoking instruction is overwritten by the original machine code. This code is an argument for the patcher and can be seen in the patcher stub at line 3. `0x41ffffff87158b4d` is the endian correct representation of a x86_64 `mov` instruction as seen in line 1 of the code snippet without patcher.

On some architectures patcher invocations may be larger than the instruction patched. This may lead to problems at basic block borders. Imagine a short instruction needing patching at the end of a basic block. The patcher invocation exceeds the basic block and overwrites some bytes in the following basic block. For correct operation the patcher would need to be executed before the following basic block is executed, but that is not assured. For that reason when a patcher is inserted into emitted code it is checked if at least `PATCHER_CALL_SIZE` bytes follow. If that is not the case an appropriate number of NOPs is inserted at the end of the basic block.

The new approach

Recent versions of CACAO use a different approach to decrease the assembler code and get rid of the patcher stubs at all. Instead of branching to a stub, an illegal instruction is generated covering the original code. The operating system raises a signal caught by a signal handler. The signal handler needs to be able to differ patchers from exceptions, e.g. by using a different trapping



```
0x00002b872a1e3d87:  4d 8b 15 a2 ff ff ff   mov    -0x5e(%rip),%r10
0x00002b872a1e3d8e:  41 ff d2               callq  *%r10
```

```
0x00002b872a1e5312:  0f 0b          ud2a
0x00002b872a1e5314:  15 87 ff ff ff  adc    $0xffffff87,%eax  (bogus)
0x00002b872a1e5319:  41 ff d2        callq  *%r10
```

```
void md_signal_handler_sigill(int sig, siginfo_t *siginfo, void *_p)
{
        /* .... */

        _uc = (ucontext_t *) _p;
        _mc = &_uc->uc_mcontext;

        /* ... */

        /* This is a patcher. */
        type = EXCEPTION_HARDWARE_PATCHER;

        /* Handle the type. */
        p = signal_handle(type, val, pv, sp, ra, xpc, _p);

        /* ... */
}
```

Figure 3.3: Patcher assembler output (new)

instruction. The handler then looks up the proper patcher and invokes it. For that purpose a list of patching positions combined with the patcher to be invoked and the data passed via patcher stack in the old approach is kept in memory. The code generator needs to provide a function called `emit_trap` capable to generate a trapping instrucion using 32 bits. Assembler code to build the patcher stack can be removed, as well as patcher stub generating code.

Figure 3.3 shows the generated assembler code on the x86_64 architecture. The illegal instruction (`u2da`) is generated where a patcher is needed When the trap instruction is executed control flow branches to a signal handler written in C. The disassembler wrongly interpretes the bytes `15 87 ff ff ff` as `adc` instruction. They are part of the offset of the `mov` instruction covered by the `ud2a` instruction.

A race condition exists when patching the trapping instruction in case he instruction can not be overwritten atomically on multiprocessor machines. One thread could just patch back the original code, while a different thread executes exactly this code and comes across a half patched instruction. For that reason single word instructions are used for trapping, as they can be written back atomically.

3.7 Exceptions

Exceptions are an integral part of the Java language and are used a lot. Nonetheless exceptions are rare events and occur irregularly. Figure 3.4 shows the number of exceptions for a run of the DaCapo benchmark suite using default data size. Although the number of exception for `lusearch` seems to be huge it has to be seen in relation to the approximately 540 million method calls.

Each method has an exception handler table associated. This table describes the start and end instruction of each exception handler directly corresponding to the Java language `try` clause. When an exception occurs at some point in the program, a lookup is performed in the exception table. The type of the occurring exception is compared to the type of each handler covering the throwing instruction.

If a match can be found the handler is executed, else the exception is propagated outside the method. For the caller this looks like a throwing invoke instruction. As the caller of a method is unknown at compile time, the caller has to be determined at runtime. This is achieved by looking up the return address which is stored on the stack. The offset is known as CACAO knows about the stack usage of each method. Stack space is allocated on method entry and no dynamic allocation is performed.

An operation called "stack unwinding" is performed whenever an exception is propagated to its caller. As control flow continues at the invoking instruction all callee saved registers have to be restored for each stack frame unwound. Callee saved register are stored on the method stack when a method is entered, therefore the restore operation is implemented by loading these registers from known stack locations.

This process either terminates when an appropriate handler has been found or the whole stack is unwound in which case the exception is unhandled and the program will be aborted.

In CACAO no explicit code is generated for calling back the runtime when an exception occurred but an illegal memory operation is performed. POSIX compatible operation systems provide a signal handling mechanism which invokes a function in this case. This signal handler tests if the memory operation was performed intentionally and if so it calls the exception handling code. In case the memory access took place unintentionally an internal exception is thrown and the vm aborts.

When native functions have been called they could have thrown an exception too. Natives can not throw exceptions directly but have to notify the runtime by setting a flag in the environment. When they return the environment is checked for an exception and exception handling code is executed when needed. Exception handling is complex because natives may call back into Java code. The stack layout is only known in JIT code, native code has a different stack layout and stack unwinding would fail when a native frame is found. Therefore a chained data structure called stackframe info is built up when invoking natives. Figure 3.5 illustrates this chaining. Technically

antlr	bloat	chart	fop	hsqldb	jython	lusearch	luindex	pdm
318	265	1194	6233	224	40040	524381	5014	15420

Figure 3.4: Number of exceptions in DaCapo

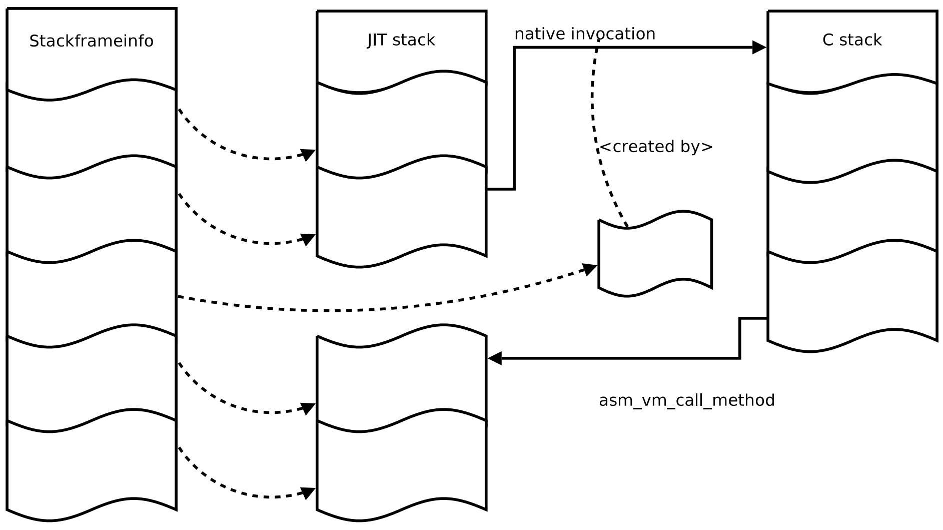


Figure 3.5: Stackframeinfo chaining with native invocation

there are no `stackframeinfo` structures for JIT frames, as this stack layout is known and contains all needed information already.

When unwinding the stack finds `asm_vm_call_method` (each native to Java invocation is performed by this function), the exception is caught and `builtin_throw_exception` is invoked which assignds the exception object to the (thread) global `exceptionptr`. When the `asm_vm_call_method` was invoked by a native the stackframeinfo structure enables the exception handling code to figure out the call site in JIT code (see `stacktrace_create`).

Chapter 4

POWERPC64 code generator

4.1 The POWERPC64 architecture

The POWERPC architecture[1] has been created in 1991 by Apple, IBM and Motorola. The specification describes many features, but a processor does not have to implement all of them. There exist specifications for 32 and 64 bit modes, as well as little and big endian modes. CACAO supports 64 bit big endian mode. POWERPC is a reduced instruction set computer (RISC) CPU, with a load-store design. It has 32 general purpose registers and 32 floating point registers. Instructions have a fixed size of 32 bit. The floating point registers are 64 bit wide and can hold IEEE-754[2] single and double precision values.

4.2 Design decisions

4.2.1 PPC64 ELF ABI

The Linux Standard Base (LSB) published by The Linux Foundation[3] specifies calling conventions and stack layout to be used for PPC64. CACAO implements LSB 3.1[4].

[1]`http://www.power.org`
[2]`http://ieee.org`
[3]`http://www.linux-foundation.org`
[4]`http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-PPC64/LSB-Core-PPC64.html`

Figure 4.1: PPC64 function descriptor

function entry point	TOC address	environment pointer

4.2.2 PPC64 CACAO ABI

The CACAO register usage is shown in figure 4.2. `r0` has a special meaning during address operations on PPC64 and can not be used uniformly therefore. To keep the register allocator simple it is marked as reserved. `r1` is used as stack pointer conforming to the ELF ABI. `r13` is reserved for system usage. `r12` is also reserved and is used by the runtime linker when invoking function descriptors.

Function descriptors

`r2` is used for linking purpose and contains a linking module specific address pointing to the start of the data segment called TOC (table of contents). The compiler generates code loading constants relative to `r2`. When invoking functions residing in an external linkage unit (e.g. a shared object) a special calling sequence invoking a so called function descriptor is used. The format of a function descriptor is shown in 4.1. The function descriptor is provided instead of the function entry point when calling an external symbol. This demands different calling conventions when linking CACAO statically.

4.2.3 Loading addresses

To load a 64 bit address into a register the assembler sequence 4.1 is needed. This is due to the 32 bit fixed sized opcodes which restrict immediate values to 16 bit for `lis` and `ori`. `rldicr` shifts the register left 32 bits and then the rest of the 64 bit value is ORed in. As even the ”Hello World” program needs about 13000 address constants using the code from listing 4.1 to load addresses would waste a lot of memory. For that reason all address constants are loaded using the data segment on PPC64. For that purpose one register is reserved and holds the start address of the data segment during execution of JIT code. This register is called `pv` the procedure vector. Loading an address constant is thus reduced to a single 32 bit long `ld` instruction with a 16 bit signed offset. For offsets beyond this 32 kb boundary the `addis` instruction is used to realize an effective offset of 32 bit which restricts the maximum size of the data segment to 2 GiB. The memory consumption is 12 byte for small offsets and 16 for larger ones instead of 20 bytes for the sequence shown in listing 4.1.

Figure 4.2: PPC64 CACAO ABI

Register	Argument	Callee saved	Caller saved	Reserved
r0				X
r1				X
r2				X
r3-r10	X			
r11				X
r12				X
r13				X
r14				X
r15		X		
r16				X
r17-r23			X	
r24-r31		X		
f0				X
f1-f13	X			
f14				X
f15				X
f16-f31		X		

```
lis     4,    address@highest
ori     4, 4, address@higher
rldicr  4, 4, 32,31
oris    4, 4, address@h
ori     4, 4, address@l
```

Listing 4.1: PowerPC64 load a 64bit address

4.2.4 Integer and long arithmetic

On 64 bit architectures implementing Java long arithmetic is easy as the machine operations directly correspond to their Java opcode counterparts. More care is needed when implementing integer arithmetic as results may exceed the value range of the integer data type. On PPC64 it is possible to implement each integer operation using long arithmetic and sign extending the result afterwards. It is important to always have integer values correctly sign extended as the `i2l` opcode (which converts integers to long) is implemented as NOP (no operation). In JIT code integer variables are treated as long, and take 8 byte on the stack when spilled. Only when reading from or writing into data structures shared with C code conversion from or to 32 bit has to be performed.

4.2.5 Float and double arithmetic

All floating point registers are 64 bit wide and can hold an IEEE-754 double precision value. It's up to the software to take care about single precision values. In CACAO the type of the register contents is always known, so it is no problem to decide whether to emit a single or double precision load/store operation. When spilling float registers on the stack `lfd` (load double) and `stfd` (store double) instruction can be used as they treat the content as bit pattern and do not convert it. Only when writing into data structures shared with C code `lfs` (load single) and `stfs` (store single) have to be used in case of single precision variables.

4.2.6 Exception handling

To trigger exception handling CACAO relies on the operating systems signal handling facilities. On PPC64 an illegal memory access (segmentation fault) is performed to trigger a signal and finally an exception. As the same mechanism is used to trigger null pointer exceptions a way to distinguish between null pointer exceptions and other exception types is needed. On PPC64 register `r0` behaves non uniform when used during address calculations where it's value is 0. This is used for triggering exceptions, as `r0` is reserved and not used during code generation. The signal handler looks up the register involved in the segmentation fault and throws a null pointer exception when `r0` was not used, but an exception of the type indicated by the offset when `r0` was used. The code listing 4.2 shows the implementation of a `CALOAD` (character array load) instruction. In line 5 a segmentation fault is triggered using `r0` as base register (denoted as 0 by the disassembler), with an offset

```
0x0000008000dd3338:   e8a30018    ld      r5,24(r3)
0x0000008000dd333c:   ea05001a    lwa     r16,24(r5)
0x0000008000dd3340:   7c358040    cmpld   r21,r16
0x0000008000dd3344:   41800008    blt-
0x0000008000dd334c
0x0000008000dd3348:   82a00002    lwz     r21,2(0)
0x0000008000dd334c:   7aac0fa4    rldicr  r12,r21,1,62
0x0000008000dd3350:   398c0020    addi    r12,r12,32
0x0000008000dd3354:   7ca5622e    lhzx    r5,r5,r12
```

Listing 4.2: PPC64 exception triggering

of 2, which corresponds to an array index out of bounds exception. The compare statement in line 3 tests the index with the size of the array loaded in line 2. When the test succeeds line 4 jumps over the faulting instruction, and program execution continues normally, otherwise the exception handling code will be triggered.

4.3 Porting PPC to PPC64

The PPC code generator was already implemented when work on the PPC64 code generator started. Due to the similarity of the two architectures most code was reused. Most work was needed to change the ABI[5]. This included a changed stack layout (8 byte slots instead of 4 byte slots). The design of a CACAO code generator has lots of subtle dependencies not expressed by the programming language. This is due to the mixing of assembler and C code as well as complexity added by the JIT only approach of CACAO. CACAO needs patchers to resolve references not known at compile time during runtime. Patchers may then load and link classes or modify offsets in assembler instructions. This leads to difficult to debug code and lots of corner cases.

An example CACAO convention: When generating an `invokevirtual` a special register (`r12`) is used to load the address of the method from the vtable of the class. This offset may not be known at JIT time, so it is possible that a patcher is created. The patcher (`patcher_invokevirtual`) resolves the method and patches back the offset. If the method needs to be compiled a call to `asm_call_jit_compiler` is generated which expects the address of the method in `r12`. After the method has been compiled

[5]`http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-PPC32/LSB-Core-PPC32/book1.html`

md_get_method_patch_address patches back the address of the newly compiled method. To differ the various cases of method invocation it inspects the opcode at the call site and in that way depends on r12 too. Additionally asm_vm_call_method needs to fake invokevirtual and therefore also depends on r12. When altering register usage any of the above functions CACAO breaks in some obscure ways.

4.4 Implementation details

Exceptions are implemented triggering segmentation violations. The lwz instruction is used for two reasons: it's the only load instruction supporting byte alignment on PPC64 and it is not used anywhere else. Byte wise alignment is needed to signal the type of exception triggered. The types are encoded as integer constants defined in exception.h. EXCEPTION_HARDWARE_ARITHMETIC e.g. has value 1 and therefore byte alignment is needed. The base register used to load from is r0 which is 0 for address operations. The segfault is handled by md_signal_handler_sigsegv. If r0 was used as base address an exception of the type encoded by the displacement is generated. Otherwise a normal null pointer exception is generated.

Patchers are generated behind the method using them. Each patcher creates a special stack frame 8 words in size. And finally invokes asm_patcher_wrapper. As invocation of asm_patcher_wrapper needs 3 assembler instructions and is the same for all patchers it is generated only once, all further patchers branch to this invocation using a single branch instruction.

asm_patcher_wrapper saves argument and temporary registers and invoked the C function patcher_wrapper providing 3 arguments, the address of the 8 word argument stack, the pv and the return address. When patcher_wrapper returns argument and temporary registers are restored (as C function may have altered them) and an exception test is performed. Depending on the result exception handling is invoked, or the patched code is executed.

patcher_wrapper reads the patcher to be invoked from 8 word argument stack, enters the Java objects monitor and attaches a stack frame for stack tracing. When the patcher returns a test for exceptions is performed and the monitor is left after the stack frame has been detached. The stack frame is important to propagate exceptions occurring while running the patcher. As patcher may initialize classes arbitrary complex Java code can be executed.

The patcher functions invoked from patcher_wrapper take the 8 word argument stack as only argument. They read all values needed from the 8 word argument stack. They then modify data segment and/or code and

Figure 4.3: PPC64 patcher argument stack

offset	content
`0`	pv of JIT method / patcher to be invoked
`1`	displacement in dseg of data to be patched
`2`	unresolved field
`3`	machine code to be patched back (on return address)
`4`	Java object in which patching takes place
`5`	return address into JIT code (to be patched)
`6`	unused on ppc64
`7`	unused on ppc64

need to invoke cache flushing functions for modified memory regions. The patchers need to be changed when the generated code is changed.

The format of the 8 word argument tack is shown in figure 4.3.

Stackwalking is generic for PPC64 as the stack is never modified and a `pv` register is used. Therefore the CACAO runtime always knows the JIT method executed (by inspecting `pv`) and can unwind the stack generic (as the stackpointer is never modified by JIT code).

Atomicity is implemented by means of `ldarx` and `stdcx` for implementing `compare_and_swap`. `ldarx` loads an value and sets a reservation. The `stdcx` instruction only succeeds when the reservation was not altered. PPC used `lwarx` and `stwcx` the 32 bit counterparts. All memory barrier related macros (`STORE_ORDER_BARRIER, MEMORY_BARRIER_AFTER_ATOMIC, MEMORY_BARRIER MEMORY_BARRIER_BEFORE_ATOMIC`) are implemented using `isync` and `sync` instructions and are identical to the PPC implementation.

Cacheflushing can be realized by the user space instructions `dcbst` and `icbi` and is implemented in `asm_cacheflush`. The cache line size is a critical parameter and is set to 128 hardcoded. This needs to be changed for chips with different cache line size (if there any).

Dynamic linking CACAO on PPC64 needs special handling. When linking statically function invocation is realized by branching to the function entry point. When linked dynamically an indirection has to be used when branching to the function entry point. As CACAO generated JIT code directly branches to the function entry point (no matter if statically or dynamically linked) not all functions have a function descriptor. Most functions in `asmpart.S` are exported as if linking would happen statically. Only functions invoked from C code need to provide a function descriptor. This are all `asm_vm_call_method*` functions and `asm_cacheflush`. These function must not be called from JIT code directly, only by means of a native stub, which is the case.

4.5 Current state and benchmark results

As the PPC64 port is rather complete it has been benchmarked intensively. All benchmarks where performed on a dual PPC970 (POWERPC G5) with 2 GHz clock speed and 1.4 GiB of RAM. The operating system used is a Gentoo GNU/Linux, kernel version 2.6.12, the compiler GCC 3.4.4 (Gentoo 3.4.4-r1, ssp-3.4.4-1.0, pie-8.7.8). The system is a pure 64 bit system. The tests were performed during night hours where the machine load is low, because it was unavailable for exclusive use. No activities of other users where noticed during the test runs. Nonetheless the results jittered a lot so each test was run 10 times, the diagrams show the best and the worst result. As reference Java implementation IBM's Java(TM) 2 Runtime Environment, Standard Edition (build pxp64dev-20060511 (SR2)) was used.

Figure 4.4 shows the results of the JVM98[6] benchmark suite. IBM's virtual machine is 50%-70% faster than CACAO, expect for _201_compress. This is probably due to the optimization features of IBM's JIT compiler[21] whereas CACAO does not do any optimizations.

Figure 4.5 shows the results of running the DaCapo[2] benchmark suite in the small configuration with 256 MiB of heap. Here the results are a bit better than in the JVM98 results with IBM's being at most 50% faster but in most tests performance is almost equal with CACAO winning 2 test, one with 70%. Nonetheless the small configuration is not recommended for real performance analysis by the DaCapo team.

Figure 4.6 shows the results of the default configuration which is recommended for performance analysis. Here the pattern of the JVM98 benchmark is repeated. CACAO is 50%-70% slower than IBM's virtual machine. Even the chart benchmark is now faster on IBM's Java, which again is probably due to runtime optimizations. Such optimizations pay off the longer the test runs as higher compile times amortize better.

To rule out performance decreasing programming errors in the PPC64 code generator the same benchmark was run on an AMD64 dual core system, with 2 GiB of RAM running at 3800+ MHz. On this platform SUN's Java implementation was benchmarked additionally. The results are shown in

[6]http://www.spec.org/jvm98/

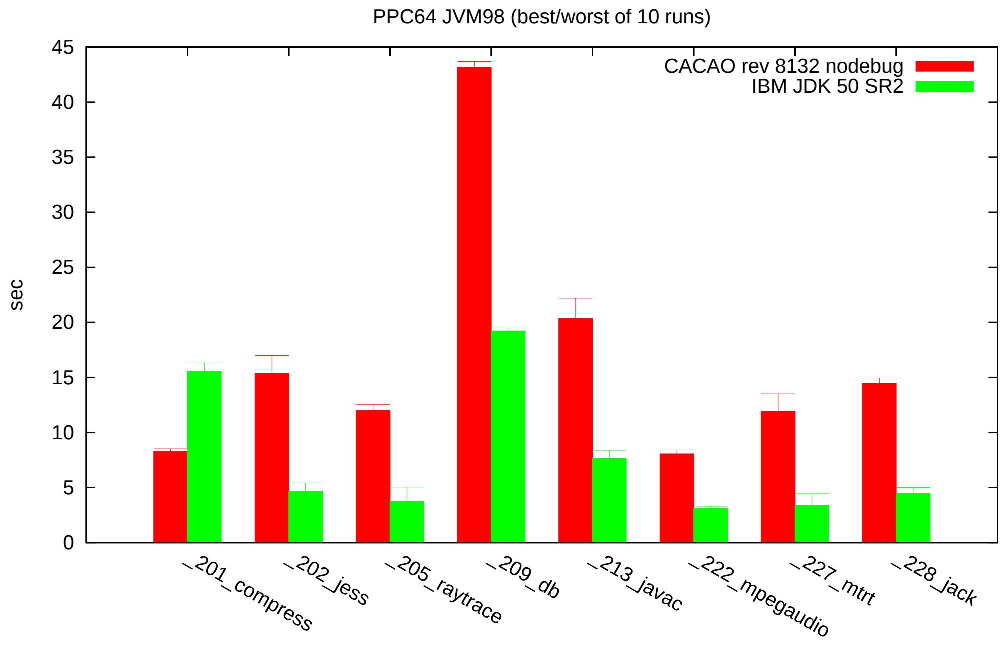


Figure 4.4: JVM98 benchmark (lower is better)

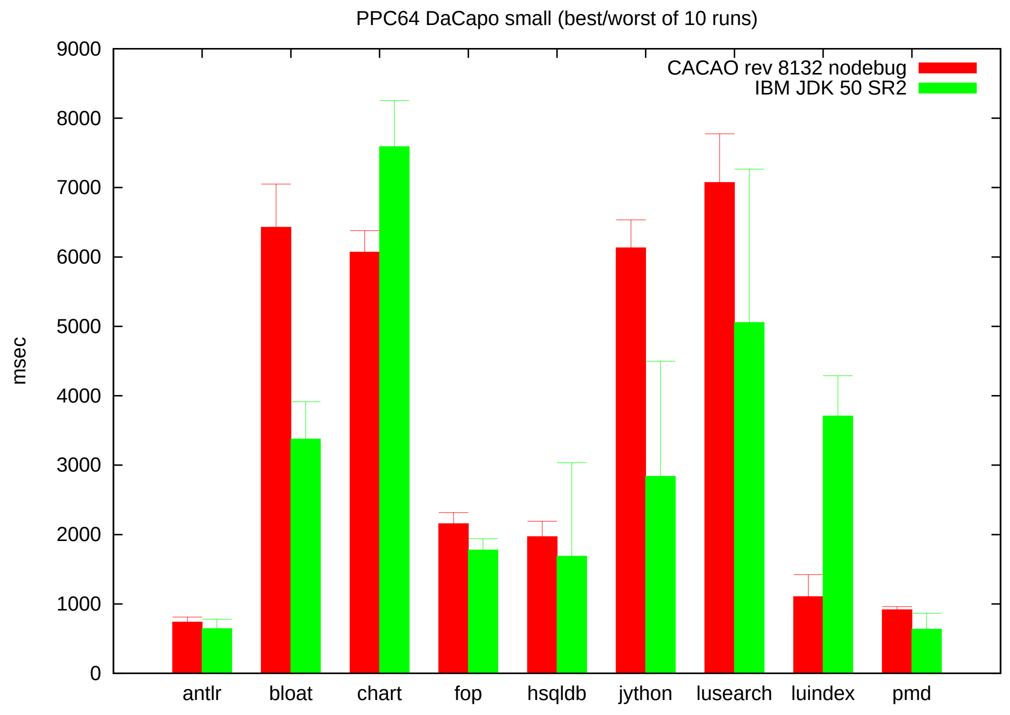


Figure 4.5: DaCapo 2006-10-MR2 benchmark (small, lower is better)

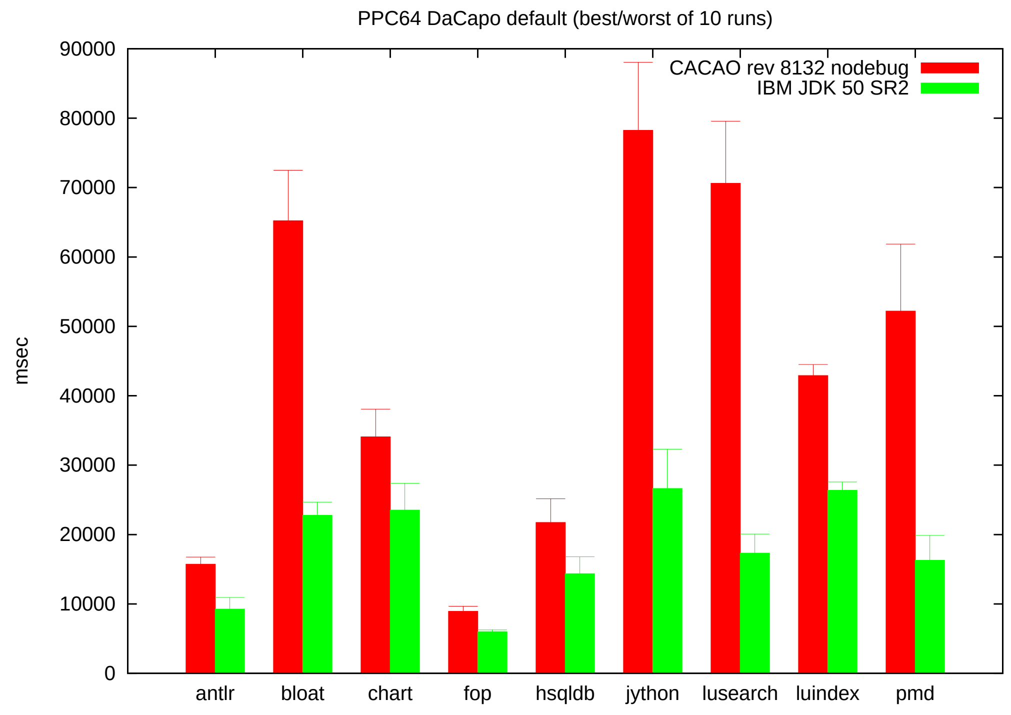


Figure 4.6: DaCapo 2006-10-MR2 benchmark (default, lower is better)

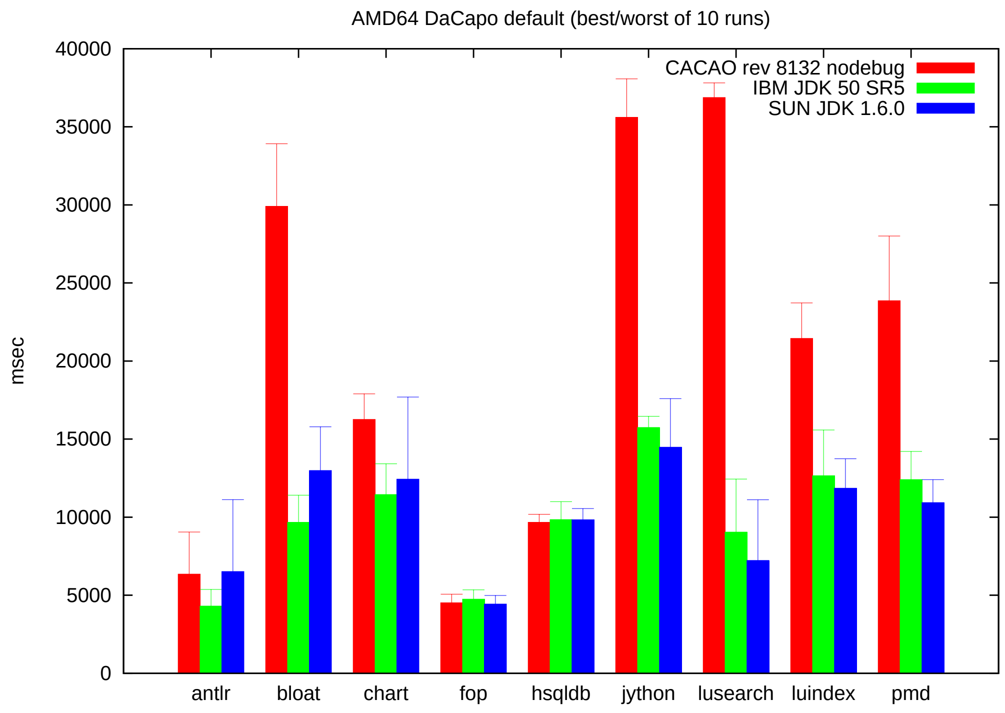


Figure 4.7: DaCapo 2006-10-MR2 benchmark (default, lower is better)

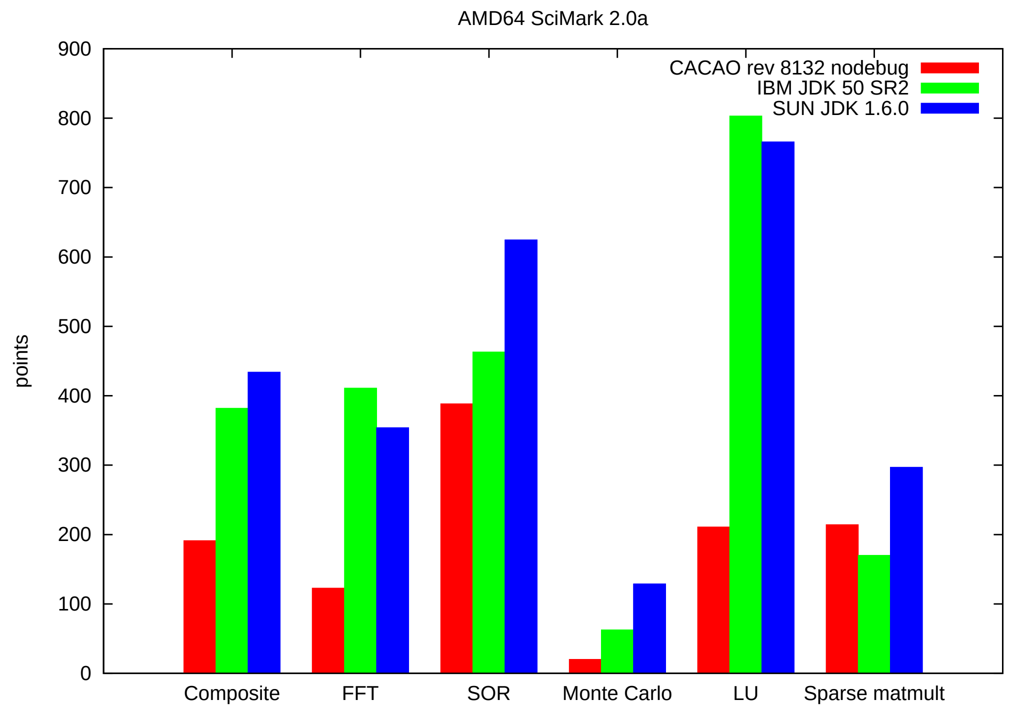


Figure 4.8: SciMark 2.0a benchmark (higher is better)

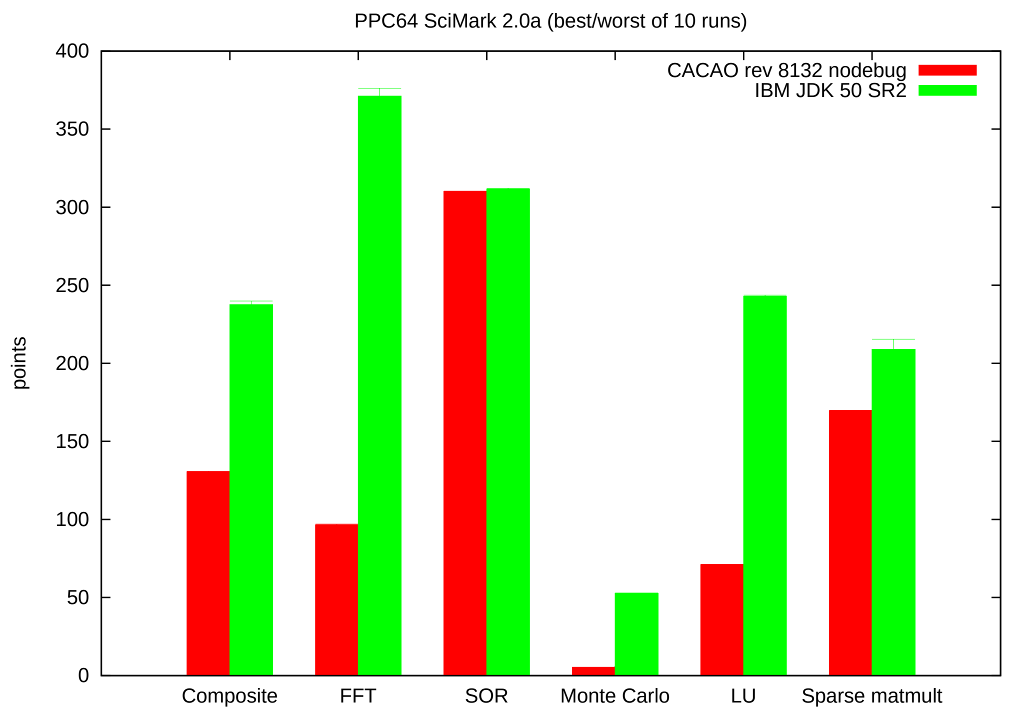


Figure 4.9: SciMark 2.0a benchmark (higher is better)

figure 4.7 and figure 4.8 and show high consistency with the results on PPC64, so the performance issues are of generic nature in CACAO.

Figure 4.9 shows the results of the SciMark[7] benchmark which mainly tests floating point operations. Even here IBM's VM is 50%-70% faster than CACAO. This is especially remarkable as those tests mainly exist of tight computational loops which indicates smart and highly specialized optimizations [21].

Figure 4.10 finally shows the results of three different CACAO versions plus IBM's VM as reference. The interesting point here is the difference between CACAO rev 8132 and CACAO rev 8132 with the `--disable-debug` configure flag and `-O2` compiler optimization flag. A 10%-30% speedup is gained by those settings.

[7] `http://math.nist.gov/scimark2/`

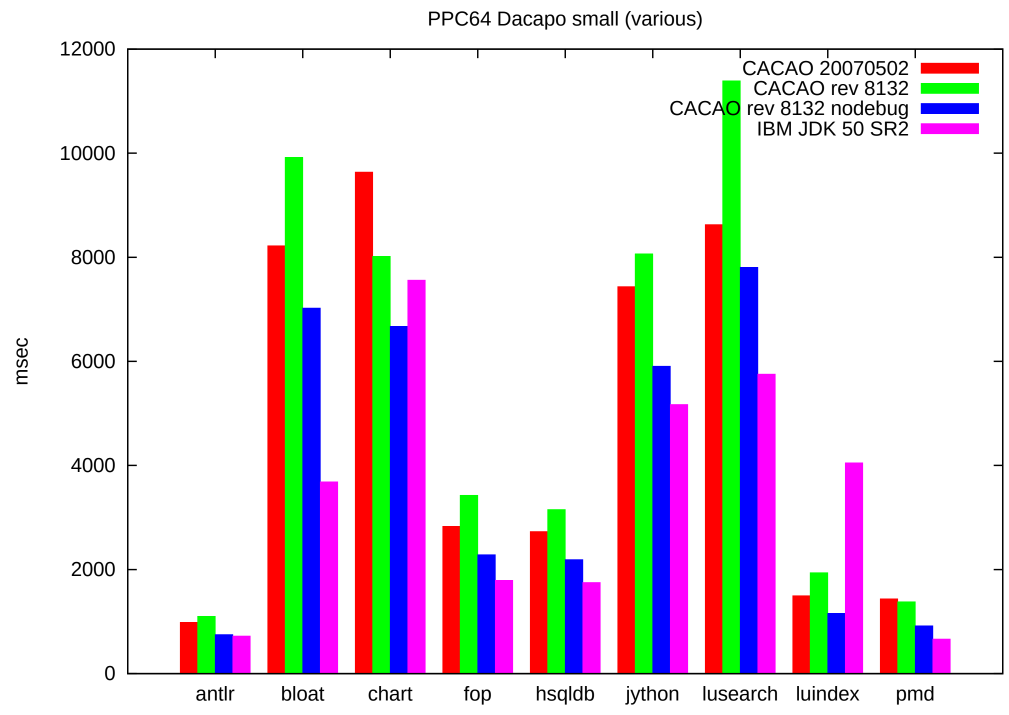


Figure 4.10: DaCapo 2006-10-MR2 benchmark (small, various, lower is better)

Chapter 5

Coldfire code generator

5.1 The Coldfire architecture

The M68K has been created 1979 by Motorola when they released the first Motorola 68000 CPU. The original M68K had 32 bit wide registers, three 16 bit ALUs which operated as a 32 bit ALU by chaining them together and a 24 bit address bus. It is a complex instruction set computer (CISC) architecture with variable sized opcodes and large immediate values. CACAO implements a code generator for Freescales Coldfire architecture[1] which implements a subset of M68K's instructions. Coldfire is a RISC architecture with 32 bit registers, a 32 bit ALU and variable sized opcodes. Versions prior to 4e did not feature a MMU or FPU, whereas later versions do so. Coldfire also defines extensions a specific CPU may implement. There exist specifications for a multiply-accumulate (MAC) and an enhanced MAC (EMAC), some earlier core versions (V2) did not support hardware division, therefore DIV is also an optional unit.

5.2 The development environment

For developing the code generator a Freescale M547xEVB LIGHT has been used. A MCF5474ZP266 Coldfire CPU was used. Due to limited power and storage a NFS root filesystem was set up and crosscompilers were used. A 2.6.10 Linux kernel was used with Freescale's Coldfire patches applied. The compiler used was a modified GCC from `http://www.codesourcery.com/`. At least version 4.1.1 should be used as version 3.4.0 has a floating point bug which is triggered by Classpath[2], the class library used by CACAO. The

[1] `http://www.freescale.com/coldfire`
[2] `http://www.classpath.org`

```
int main(int args, char**argv)
{
  float f = 3.14;
  double d = 3.14;

  if ( f == (float)d )    {
    printf("Foo\n");
  } else  {
    printf("Bar\n");
  }

  f = 0.75;
  d = 0.75;

  if ( f == (float)d )    {
    printf("Foo\n");
  } else  {
    printf("Bar\n");
  }
}
```

Listing 5.1: Coldfire compiler bug testcase

test case extracted is shown in listing 5.1. The output of the two programs compiled with different compiler versions is given in listing 5.2. The problem is the conversion from double to float nested in an if statement. In this case the conversion is not done. As the test case illustrates not all double values have different encoding as float and therefore the bug is not triggered every time.

Because CACAO modifies code during runtime instruction caches need to be flushed after code has been modified or subtle non-deterministic bugs may show up. Coldfire has a cache flush operation (CPUSHL) but this operation may only be executed in supervisor mode of the CPU. As CACAO does not run native but depends on an operating system supervisor instructions can not be executed directly (CACAO is executed in user mode). Therefore the operation system has to provide a system call which executes cache flushing operations for the application if needed. The Coldfire patches for Linux 2.6.10 only provided a cache flush system call for original M68K and not for Coldfire so a patch was developed. The `sys_cacheflush` function is implemented in `arch/m68k/kernel/sys_m68k` and has to be modified as given in listing 5.3. It has to be noted that this patch implements cache flush-

```
/ # ./test411
Foo
Foo
/ # ./test340
Bar
Foo
/ #
```

Listing 5.2: Coldfire compiler bug testrun

ing for Coldfire only, the code for M68K cache flushing has been removed. At present both caches are flushed, regardless of the `scope` parameter which enables the application to indicate which cache to flush (data or instruction). `DcacheFlushInvalidateCacheBlock` and `IcacheInvalidateCacheBlock` are part of the Freescale patches and violate Linux kernel naming conventions[3].

5.3 Design decisions

5.3.1 CACAO register usage

Coldfire CPUs differ integer registers, address registers and floating point registers. All floating point registers are wide enough to hold double precision values but can also operate on single precision values. There are 8 registers of each type available. CACAO reserves 3 registers as temporary registers which are used exclusively for implementing byte code instructions, however not all instructions need all 3 registers. For the Coldfire port it was decided to reserve 3 registers of each type as temporary registers. This made implementing instructions easy as one can move data from integer to address registers when the register pressure is high, but wastes registers. A clean solution would be a register allocator for temporary registers. As such a feature is planned no effort was made to reduce the number of reserved registers, although the benchmarking results show a 15% speedup increasing the number of CACAO used address register (see figure 5.4). Likely a noticeable speedup could be achieved by reducing the number of temporary registers.

[3]Documentation/CodingStyle in the Linux source tree

```
/*sys_cacheflush--flush (part of) the processor cache.*/
asmlinkage int
sys_cacheflush(unsigned long addr, int scope, int cache,
               unsigned long len)
{
  struct vm_area_struct *vma;

  lock_kernel();

  vma = find_vma (current->mm, addr);
  if (vma == NULL || addr < vma->vm_start ||
      addr + len > vma->vm_end) {
    printk("sys_cacheflush␣parameters␣invalid");
    if (vma != NULL)
      printk("vma=%p,start=%x,end=%x,addr=%x,len=%d\n",
              vma,vma->vm_start,vma->vm_end,addr,len);
      unlock_kernel();
      return -1;
  }

  DcacheFlushInvalidateCacheBlock(addr, len);
  IcacheInvalidateCacheBlock(addr, len);

  unlock_kernel();
  return 0;
}
```

Listing 5.3: Linux 2.6.10 sys_cacheflush patch

Figure 5.1: M68K stack layout

Offset	Content
`%fp` + 0	previous framepointer
`%fp` + 4	return address
`%fp` + 8	arguments

5.3.2 M68K ABI

The application binary interface (ABI) of Coldfire is the same as for M68K, but no real documentation exists online. The following calling conventions are used by the Gnu Compiler Collection (GCC) and CACAO follows these conventions when calling external functions.

- Arguments are passed via stack. There are no register arguments. All arguments are passed using 32 bit slot size. Double and long arguments use 2 stack slots.
- Arguments are pushed onto the stack, the last argument as pushed first.
- When returning a value register `%d0` is used when the return value is 64 bit `%d1` is used in addition. [4]
- A callee may destroy the contents of the registers `%d0`, `%d1`, `%a0`, `%a1`, `%f0` and `%f1`.
- All other registers are callee saved.

Register `%a7` (`%sp`) is used as stack pointer and `%a6` (`%fp`) as framepointer. Stack and framepointer are chained using the `link` and `unlk` assembler instructions. The stack layout used by GCC is shown in table 5.1. There are however no guarantees, as no official documentation exists, and the stack layout is know to be violated when compiling code with the `-fomit-frame-pointer` flag.

5.3.3 CACAO ABI

For CACAO it was decided to stick to the C ABI as close as reasonable. However, some things were changed. The CACAO register usage is shown in

[4]There are however cases when GCC expects pointer return value in `%a0`, whether this is a bug or not was not further investigated, CACAO as a workaround returns in both `%a0` and `%d0`.

Figure 5.2: M68K CACAO ABI

Register	Argument	Callee saved	Caller saved	Reserved
`%a0`			X	
`%a1`			X	
`%a2`				X
`%a3`				X
`%a4`				X
`%a5`		X		
`%a6`		X		
`%a7`				X
`%d0`			X	
`%d1`			X	
`%d2`				X
`%d3`				X
`%d4`				X
`%d5`		X		
`%d6`		X		
`%d7`		X		
`%f0`			X	
`%f1`			X	
`%f2`				X
`%f3`				X
`%f4`				X
`%f5`		X		
`%f6`		X		
`%f7`		X		

table 5.2. Register `%a7` (`%sp`) is used as stack pointer and therefore reserved, `%a6` which is used as framepointer (`%fp`) in the C ABI is used as additional register in JIT code as stack frame chaining would be redundant, as the stack usage of each method is known a priori. Method arguments are passed via stack solely, no argument registers are used.

5.3.4 Dedicated `pv` register

As Coldfire does not have a huge amount of available registers it was decided not to use a dedicated `pv` register. The `pv` is the so called procedure vector, a pointer to the entry of the JIT method. It is used to access the data segment which is below method's code. As M68K supports 32 bit immediate values for load operations it was decided not to use the data segment at all for code

```
0x40b0280a:   246f 0000        moveal %sp@(0),%a2
0x40b0280e:   4a8a             tstl %a2
0x40b02810:   6602             bnes 0x40b02814
0x40b02812:   4e4e             trap #14
0x40b02814:   247c 40af a0cc   moveal #1085251788,%a2
0x40b0281a:   4e92             jsr %a2@
```

Listing 5.4: M68K exception using trap

generation, but emit opcodes using immediate values.

5.3.5 Branching instructions

There is a 16 and 8 bit program counter relative address mode which could be use for size optimization. The 16 bit variant is used during code generation, the 8 bit variant is currently not used. For the antlr benchmark of the DaCapo[2] benchmark suite (version 2006_10_MR2) on M68K a total number of 13322 branch instruction are generated of which 7585 have 8 bit offsets. This are around 57% of all branches. Starting eclipse [5] on PPC64 generated 75438 branches of which 50111 (66%) have 8 bit offset. So introducing 8 bit offset optimization could save around 400k of code for eclipse. On the other hand 20M of memory (generated code and data, no CACAO runtime) is used in total. So the overall saving would be approximately 2%. It has to be noted however, that 8 bit branching offsets are used implementing the complex byte instructions like e.g. `instanceof` and `checkcast`.

5.3.6 Exceptions

Coldfire's instruction set includes a trap instruction with a 4 bit vector argument. This instruction is translated into an illegal instruction signal (`SIGILL`) by the Linux kernel. A signal handler is installed by CACAO which inspects the trapping instruction. If the `trap` instruction triggered the signal exception handling is initiated, else an error is reported and the VM aborts. An example of this mechanism is shown in listing 5.4. This is the disassembled machine code of an `INVOKESPECIAL` instruction. The value in `%a2` is tested against 0. If not equal the trap instruction is jumped over otherwise the trap is executed and an exception will be thrown. The value `14` denotes the exact type of the exception (a null pointer exception in this case).

[5]http://eclipse.org

```
0x40b02f50:   2f48 0000         movel %a0,%sp@(0)
0x40b02f54:   246f 0000         moveal %sp@(0),%a2
0x40b02f58:   266a 0000         moveal %a2@(0),%a3
0x40b02f5c:   286b 005c         moveal %a3@(92),%a4
0x40b02f60:   4e94              jsr %a4@
```

Listing 5.5: M68K exception by illegal memory access

Nullpointer exceptions can also be triggered when invoking a method on a `null` object. To spare the code needed to test each object reference before invoking a method CACAO relies on the memory management unit (MMU) of the Coldfire CPU. As example the code generated for a `INVOKEVIRTUAL` instruction is shown in listing 5.5. The object is originally stored in `%a0`. In line 3 the vtable of the object is accessed and finally, in line 4, the address of the method to be invoked is loaded. When the object was the null object (and therefore %a2 contained value 0), line 3 will perform an illegal memory access and a signal (SIGSEGV) will be generated by the kernel. CACAO catches the signal and throws an exception (a null pointer exception in this case).

5.3.7 Native double return values

Because the M68K ABI declares %d0 and %d1 as return registers for 64 bit values double values are returned via these two registers as well. The Coldfire instruction set does not offer an instruction to load a floating point register from two integer registers, so both integer registers have to be written into memory and read back as double value. This demands two additional stack slots to be allocated for each method, as it is unknown whether a native function will be invoked or not. Synchronized Java methods need a lock word, which resides on the method stack as well. This lock word is normally only allocated when the method has the synchronized flag set. On Coldfire the lock word shares one stack slot with the two slots needed for the double conversion as their lifetimes do not overlap, so the code generator allocates two additional stack slots for the (unlikely) conversion, but does not need to allocate a stack slot in case of synchronized methods.

5.4 Implementation details

Exceptions are triggered using the `trap` instruction. A difficulty arises for `EXCEPTION_HARDWARE_NULLPOINTER` which is defined 0 in `exceptionis.h`.

Figure 5.3: M68K patcher argument stack

offset	content
`0`	patcher to be invoked
`1`	unresolved field
`2`	2nd machine word to be patched back
`3`	machine word to be patched back
`4`	Java object
`5`	REG_ITMP3 which needs to be restored

The `trap` instruction with argument 0 is caught inside the kernel and never delivered to userspace. A special symbol named `M68K_EXCEPTION_HARDWARE_NULLPOINTER` is defined to 14 and used instead of the original define. An additional signal handler `md_signal_handler_sigill` is used to catch the signals triggered by `trap`. For classcast checks and array index out of bounds checks an additional argument is needed for the exception handling mechanism (the object, the index). As the `trap` instruction can not be used to encode more data an instruction with negligible side effect is generated preceding the `trap`. The `tst.b` instruction not used anywhere else in JIT code has been chosen. It takes an register as argument, the one containing the value of interest. The `M_TRAP_SETREGISTER` macro is used to generate the instruction.

Patchers are generated after the method using them. In contrast to the PPC64 solution each patcher directly branches to `asm_patcher_wrapper`. The argument stack size is 6 words and the layout shown in figure 5.3.

Stackwalking needs special handling on Coldfire as neither a `pv` register exists nor is the stack pointer exclusively controlled by CACAO. CACAO runtime provides `codegen_get_pv_from_pc` which is used to calculate `pv` when needed. This function queries an AVL tree[5] with knowledge about all generated methods to calculate the `pv` from an address part of the method. The `jsr` instruction (jump subroutine) used to invoked function implicitly pushes the return address onto the stack. When unwinding the stack this offset has to be considered. The CACAO runtime supports this case, it is enabled by adding M68K defines in `stacktrace.c` and `stacktrace.h`. When removing the stackframe the size is increased by one word to consider the return address.

Atomicity can not be achieved on Coldfire as the hardware lacks support. `compare_and_swap` is implemented but not atomically. A compiler warning is generated to warn users about this fact. The memory barrier related macros are not implemented as well.

5.5 Current state and benchmark results

The code generator developed is complete and reasonable bug free, but as CACAO lacks an own garbage collector (GC) and the Boehm GC [6] caused segmentation faults only a small number of tests could be run. Because an exact garbage collector had been in development while implementing the code generator it was decided not to fix Boehm GC on Coldfire. Testing has to be resumed as soon as the garbage collector is completed. The M68K port successfully runs jctest, a CACAO internal test program, fptest (same for floating point arithmetic), DaCapo antlr, SciMark and many other programs with small memory footprint.

The following benchmarks were performed with different features of the code generator enabled or disabled. In the graphs "CACAO" refers to a CACAO build with all features enabled. In earlier versions of the code generator the ABI reserved register %fp was not used within JIT code, "CACAO without fp" refers such a build, but is otherwise identically to CACAO. The M68K code generator may be compiled with soft floating point arithmetic which does not use floating point registers and instructions, this build is called "CACAO with softfloat". The benchmarks were produced using CACAO rev 8100 running on a ColdFire V4e @ 131.4 MHz with 64 MB RAM, a NFS mounted root file system under Linux Freescale 2.6.10 kernel. The whole system has been crosscompiled using the m68k-linux-gnu-gcc (CodeSourcery Sourcery G++ 4.1-30) 4.1.1 compiler.

DaCapo antlr

The DaCapo[2] antlr benchmark is a parser generator which reads in a grammar file and generates a parser and lexical analyzer for it. It rarely uses floating point arithmetic. Figure 5.4 shows how the 3 versions of CACAO perform with this load. The most interesting case is the difference between "CACAO" and "CACAO without fp", a 16.5% speedup. This speedup is solely caused by using one additional address register.

[6]`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

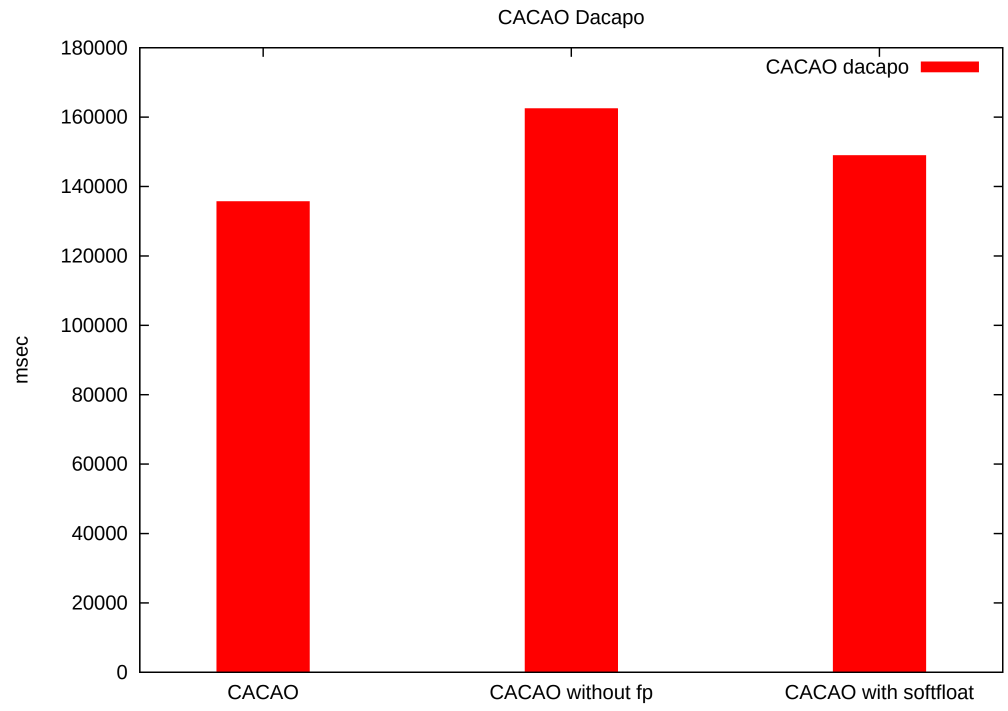


Figure 5.4: DaCapo 2006-10-MR2 antlr benchmark

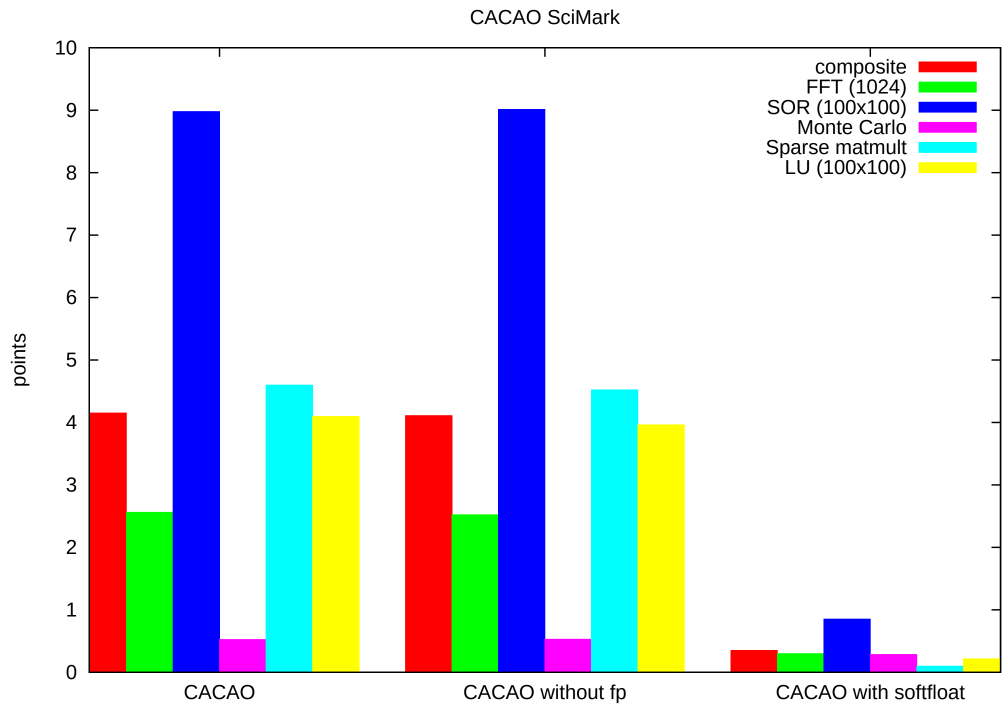


Figure 5.5: SciMark 2.0a benchmark

Chapter 6

Summary

6.1 Conclusion

The aim of this work was to develop CACAO code generators for POWERPC64 and Coldfire. The POWERPC64 port is officially part of the 0.98 release of CACAO and passes all internally used test cases. It can also be used to run well known Java programs like Eclipse[1] and Jboss[2]. Bugs left will be fixed by the CACAO team when reported.

The Coldfire port is not as mature as POWERPC64. The main problem are limited memory resources of the development board and the unfinished exact garbage collector. Nonetheless the port is complete which means that all features are implemented, what remains are bugs which can be fixed as soon as the garbage collector works on Coldfire. The first steps for implementing the garbage collector have been done and will be finished soon.

6.2 Future

Future plans for the code generators include a patching mechanism using signals instead of assembler stubs, as well as JIT compiler invocation via signals. These demands became clear when implementing the exact garbage collector. This will reduce the assembler language usage and improve maintainability, a problem encountered during PPC64 porting.

The register allocator could be enhanced to reduce the number of temporary registers permanently reserved for implementing byte code instructions. Most instruction do not need 3 temporary registers. The situation is even worse for architectures with designated address registers like M68K where 6

[1]`http://www.eclipse.org/`

[2]`http://www.jboss.org/`

registers are reserved. One problem to solve is the need of a register usage description per byte code instruction, which may be architecture specific.

There are plans to implement the exception handling code in C language. This would further reduce the lines of assembler code needed to implement a code generator. Exception handling code is among the most difficult parts for programmers, so porting CACAO for a new platform would become much easier.

During this work a change in `asm_vm_call_method` was introduced. Originally this function was called with an array containing value and type of the arguments for the Java method to be invoked. The function then transformed this data structure to the platform ABI. On many platforms this included multiple loops and difficult offset calculations for stack arguments. Especially POWERPC64 has a very difficult ABI to implement. The new arguments consist of two arrays, one containing the values for all argument registers, the second containing the stack for the callee. The function now simply iterates both arrays and copies the arguments into registers and onto the stack. The stack has been prepared in C code which is much easier than in assembler language. This patch saved around 300 lines of code on POWERPC64 and made `asm_vm_call_method` almost trivial. This changes need to be ported to all code generators.

6.3 Source code

All source code developed for this master thesis has been committed to a public accessibly Mercurial[3] repository. Anonymous access is enabled. On an UNIX system the following command checks out the source code.

```
$hg clone http://mips.complang.tuwien.ac.at/hg/cacao/ cacao
```

The code generators reside in the directories `src/vm/jit/powerpc64` and `src/vm/jit/m68k` and have been put under the GPL license. More information is available from `http://cacaojvm.org`.

[3] `http://www.selenic.com/mercurial/wiki/`

Bibliography

[1] Alpern Augart Blackburn. The jikes research virtual machine project: Building an open-source research community. *IBM SYSTEMS JOURNAL*, 44(2), 2005.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, October 2006. ACM Press.

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

[4] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel®itanium^{TM}processor. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 182–191, Washington, DC, USA, 2001. IEEE Computer Society.

[5] C. C. Foster. Information retrieval: information storage and retrieval using avl trees. In *Proceedings of the 1965 20th national conference*, pages 192–205, New York, NY, USA, 1965. ACM Press.

[6] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[7] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.*, 2(1-4):135–150, 1993.

[8] Jason Hong. The use of java as an introductory programming language. *Crossroads*, 4(4):8–13, 1998.

[9] Xianglong Huang, Stephen M. Blackburn, David Grove, and Kathryn S. McKinley. Fast and efficient partial code reordering: taking advantage of dynamic recompilatior. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, pages 184–192, New York, NY, USA, 2006. ACM Press.

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

[11] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in java programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.

[12] David Holmes Ken Arnold, James Gosling. *The Java Programming Language.* Addison-Wesley Longman, 2005.

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

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

[15] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[16] Todd A. Proebsting. Burs automata generation. *ACM Trans. Program. Lang. Syst.*, 17(3):461–486, 1995.

[17] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M.

Levy. The structure and performance of interpreters. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 150–159, New York, NY, USA, 1996. ACM Press.

[18] Papalaskaris M. A. Schubert L. K. and J Taugher. Determining type, part, color, and time relationships. *IEEE Computer (special issue on Knowledge Representation)*, 16(10), 1983.

[19] Edwin Steiner. Adaptive inlining via on-stack replacement. Diplomarbeit, Vienna University of Technology, 2007.

[20] Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 172–181, Washington, DC, USA, 2001. IEEE Computer Society.

[21] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.

[22] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM Press.

[23] Ankush Varma and Shuvra S. Bhattacharyya. Java-through-c compilation: An enabling technology for java in embedded systems. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 30161, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM Press.