DIPLOMARBEIT

# Optimizing and Porting the CACAO JVM

ausgeführt am

Institut für Computersprachen
Programmiersprachen und Übersetzer
der Technischen Universität Wien

unter der Anleitung von

A.o.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

durch

Christian Thalinger
Myrthengasse 9/16 1070 Wien

Wien, am 30. August 2004

## Abstract

The CACAO Java Virtual Machine was designed as a 64-bit Java Virtual Machine on the Institut für Computersprachen der Technischen Universität Wien in 1996. The primary intent of CACAO was to build the fastest Just-In-Time compiler available at this time of the Alpha architecture.

This document describes some optimizations and enhancements implemented in CACAO like lazy class loading and instruction combining with constant operands to speed up compile and run time.

Furthermore experiences of porting the CACAO Java Virtual Machine to two new architectures are presented. These new architectures are the IA32 and AMD64 architecture. Especially the porting to the IA32 architecture is interesting because a 32-bit architecture is not a preferred target architecture for CACAO.

Finally the implemented optimizations and the ports are evaluated against CACAO itself and well-known Java Virtual Machines.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 About CACAO

The CACAO Java Virtual Machine [Gra97] was designed as a 64-bit Java Virtual Machine on the Institut für Computersprachen der Technischen Universiät Wien in 1996. A Java Virtual Machine [LY99] executes Java byte code which is the binary representation of a compiled source code written in the Java Programming Language [GJSB00].

The primary intent of CACAO was a fast compiling and executing Java Virtual Machine, thus is was designed to use a *compile-only* approach with a Just-In-Time compiler. *Compile-only* means that the Java Virtual Machine only uses a compiler which produces native machine code and no interpreter. A Just-In-Time compiler compiles each executed Java method into native machine code exactly when the method is called.

The target architecture CACAO was developed for is the DEC Alpha. The Alpha architecture is a 64-bit RISC architecture which means that CACAO can be easily ported to other 64-bit RISC architectures like MIPS.

## 1.2 Motivation

The early CACAO implementation was trimmed to run some benchmarks and daily-usage console Java programs like Java compilers. This means a simple thread support was implemented and a mostly static run time system. This implementation was sufficient to run the Sun Java compiler `javac` and most SPEC benchmarks [Cor98] which have been patched to run as console applications since AWT was not implemented. The development of CACAO has nearly stopped in 1999.

In 2002 the CACAO development team decided to push the CACAO development further and to port CACAO to the famous IA32 architecture. After this port was done the next obvious architecture was AMD's new AMD64 architecture. The experiences of both ports are described throughout this document in section 3.6 and 3.7 respectively.

Since CACAO was now ready to nearly run on every PC, aside from the fact that CACAO does not compile or run under Microsoft Windows yet, the CACAO development team decided to make CACAO a standard compliant Java Virtual Machine and to release CACAO under the GNU General Public License in the future. This means that some run time parts of CACAO had to be restructured mostly to be thread safe and reentrant.

One big part was the class loading system. A standard compliant Java Virtual Machine must support lazy class loading and lazy class linking. Since CACAO only supported eager class loading and linking this has to be implemented completely new (see chapter 2). This change not even made CACAO more standard compliant but made it faster in startup times of the Java Virtual Machine which will be shown in the evaluation chapter of this document (see chapter 4).

# Chapter 2

# Class Loader

## 2.1 Introduction

A Java Virtual Machine dynamically loads, links and initializes classes and interfaces when they are needed. Loading a class or interface means locating the binary representation—the class files—and creating a class or interface structure from that binary representation. Linking takes a loaded class or interface and transfers it into the runtime state of the Java Virtual Machine so that it can be executed. Initialization of a class or interface means executing the static class or interface initializer `<clinit>`.

The following sections describe the process of loading, linking and initalizing a class or interface in the CACAO Java Virtual Machine in greater detail. Further the used data structures and techniques used in CACAO and the interaction with the GNU Classpath [Fou04] are described.

## 2.2 System class loader

The class loader of a Java Virtual Machine is responsible for loading all type of classes and interfaces into the runtime system of the Java Virtual Machine. Every Java Virtual Machine has a *system class loader* which is implemented in `java.lang.ClassLoader` and this class interacts via native function calls with the Java Virtual Machine itself.

The GNU Classpath implements the system class loader in the GNU Classpath specific class `gnu.java.lang.SystemClassLoader` which extends `java.lang.ClassLoader` and interacts with the Java Virtual Machine. The *bootstrap class loader* is implemented in `java.lang.ClassLoader` plus the Java Virtual Machine depended class `java.lang.VMClassLoader`. This class is the main class how the bootstrap class loader of the GNU Classpath interacts with the Java Virtual Machine. The main function of this class is

```
static final native Class loadClass(String name, boolean resolve)
  throws ClassNotFoundException;
```

This native function is implemented in the CACAO Java Virtual Machine, located in `nat/VMClassLoader.c` and calls the internal loader functions of CACAO class loading system. If the `name` argument is `NULL`, a `java.lang.NullPointerException` is created and the function returns `NULL`.

If the `name` is non-NULL a new UTF8 string of the class' name is created in the internal *symbol table* via

```
utf *javastring_toutf(java_lang_String *string, bool isclassname);
```

This function converts a `java.lang.String` string into the internal used UTF8 string representation. `isclassname` tells the function to convert any `.` (periods) found in the class name into `/` (slashes), so the class loader can find the specified class in the file system or in the `zip`/`jar` file.

Then a new `classinfo` structure (see figure 2.1) is created via the

```
classinfo *class_new(utf *classname);
```

function call. This function creates a unique representation of this class, identified by its `classname`, in the JVM's internal *class hashtable*. The newly created `classinfo` structure is initialized with default values like setting `loaded`, `linked` and `initialized` to `false` which guarantee a definite state of a new class.

The next step is to actually load the class requested. Thus the main loader function

```
classinfo *class_load(classinfo *c);
```

is called, which is a wrapper function to the real loader function

```
classinfo *class_load_intern(classbuffer *cb);
```

```
struct classinfo {                /* class structure                        */
    ...
    s4          flags;            /* ACC flags                              */
    utf        *name;             /* class name                             */

    s4          cpcount;          /* number of entries in constant pool     */
    u1         *cptags;           /* constant pool tags                     */
    voidptr    *cpinfos;          /* pointer to constant pool info structures */

    classinfo  *super;            /* super class pointer                    */
    classinfo  *sub;              /* sub class pointer                      */
    classinfo  *nextsub;          /* pointer to next class in sub class list */

    s4          interfacescount;  /* number of interfaces                   */
    classinfo **interfaces;       /* pointer to interfaces                  */

    s4          fieldscount;      /* number of fields                       */
    fieldinfo  *fields;           /* field table                            */

    s4          methodscount;     /* number of methods                      */
    methodinfo *methods;          /* method table                           */
    ...
    bool        initialized;      /* true, if class already initialized     */
    bool        initializing;     /* flag for the compiler                  */
    bool        loaded;           /* true, if class already loaded          */
    bool        linked;           /* true, if class already linked          */
    s4          index;            /* hierarchy depth (classes) or index     */
                                  /* (interfaces)                           */
    s4          instancesize;     /* size of an instance of this class      */
    ...
    vftbl_t    *vftbl;            /* pointer to virtual function table      */

    methodinfo *finalizer;        /* finalizer method                       */

    u2          innerclasscount;  /* number of inner classes                */
    innerclassinfo *innerclass;
    ...
    utf        *packagename;      /* full name of the package               */
    utf        *sourcefile;       /* classfile name containing this class   */
    java_objectheader *classloader; /* NULL for bootstrap classloader       */
};
```

Figure 2.1: `classinfo` structure

This wrapper function is required to ensure some requirements:

- enter a monitor on the `classinfo` structure to make sure that only one thread can load the same class or interface at the same time
- check if the class or interface is `loaded`, if it is `true`, leave the monitor and return immediately
- measure the loading time if requested
- initialize the `classbuffer` structure with the actual class or interface data
- reset the `loaded` field of the `classinfo` structure to `false` and remove the `classinfo` structure from the internal class hashtable if an error or exception during loading
- free any allocated memory
- leave the monitor

The `class_load` function is implemented to be *reentrant*. This must be the case for the *eager class loading* algorithm implemented in CACAO (described in more detail in section 2.3.1). Furthermore this means that serveral threads can load different classes or interfaces at the same time on multiprocessor machines.

The `class_load_intern` function preforms the actual loading of the binary representation (see figure 2.2) of the class or interface.

During loading some verifier checks are performed which can throw an error. This error can include a `java.lang.ClassFormatError` or a `java.lang.NoClassDefFoundEr`
Some of these `java.lang.ClassFormatError` checks are

- *Truncated class file* — unexpected end of class or interface data
- *Bad magic number* — class or interface does not start with the magic bytes (`0xCAFEBABE`)
- *Unsupported major.minor version* — the byte code version of the given class or interface is not supported by the JVM

The actual loading of the bytes from the binary representation is done via the `suck_*` functions. These functions are

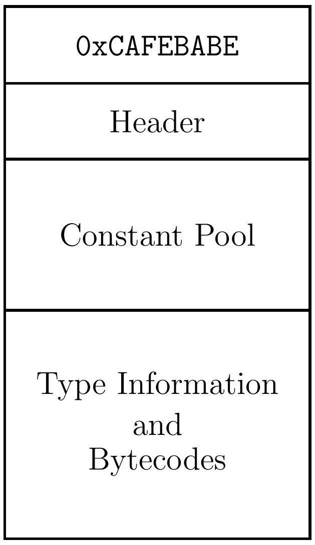

Figure 2.2: Binary representation of a class or interface

- `suck_u1`: load one `unsigned byte` (8 bit)
- `suck_u2`: load two `unsigned bytes` (16 bit)
- `suck_u4`: load four `unsigned bytes` (32 bit)
- `suck_u8`: load eight `unsigned bytes` (64 bit)
- `suck_float`: load four `bytes` (32 bit) converted into a `float` value
- `suck_double`: load eight `bytes` (64 bit) converted into a `double` value
- `suck_nbytes`: load $n$ bytes

Loading `signed` values is done via the `suck_s[1,2,4,8]` macros which cast the loaded `unsigned bytes` to `signed` values. All these functions take a `classbuffer` structure pointer as argument which contains the binary representation and the size of the class or interface.

This `classbuffer` structure is filled with data via the

```
classbuffer *suck_start(classinfo *c);
```

function. This function tries to locate the class, specifed with the `classinfo` structure, in the `CLASSPATH`. This can be a plain class file in the file system or a file in a `zip/jar` file. If the class file is found, the `classbuffer` is filled with data collected from the class file, including the class file size and the binary representation of the class.

Before any read of a byte from the binary representation with a `suck_*` function, the remaining bytes in the `classbuffer` data array must be checked with the

```
static inline bool check_classbuffer_size(classbuffer *cb, s4 len);
```

function. If the remaining bytes number is less than the amount of the bytes to be read, specified by the `len` argument, a `java.lang.ClassFormatError` with the detail message *Truncated class file*—as mentioned before—is thrown.

The following subsections describe chronologically in greater detail the individual loading steps of a class or interface from it's binary representation.

## 2.2.1 Constant pool loading

The class' constant pool is loaded via

```
static bool class_loadcpool(classbuffer *cb, classinfo *c);
```

from the `constant_pool` table in the binary representation of the class of interface. The `classinfo` structure has two pointers to arrays which contain the class' constant pool infos, namely: `cptags` and `cpinfos`. `cptags` contains the type of the constant pool entry. `cpinfos` contains a pointer to the constant pool entry itself.

The constant pool needs to be parsed in two passes. But some constant pool types can be completely resolved in the first pass and need no further processing. These types are:

- `CONSTANT_Integer`
- `CONSTANT_Float`
- `CONSTANT_Long`
- `CONSTANT_Double`
- `CONSTANT_Utf8`

The remaining constant pool entries need two passes:

- `CONSTANT_Class`
- `CONSTANT_String`
- `CONSTANT_NameAndType`
- `CONSTANT_Fieldref`
- `CONSTANT_Methodref`
- `CONSTANT_InterfaceMethodref`

In the first pass the information loaded is saved in temporary structures, which are further processed—if required—in the second pass, when the complete constant pool has been traversed. Only when all constant pool entries have been processed, every constant pool entry can be completely resolved, but this resolving can only be done in a specific order:

1. `CONSTANT_Class`
2. `CONSTANT_String`
3. `CONSTANT_NameAndType`
4. `CONSTANT_Fieldref`
   `CONSTANT_Methodref`
   `CONSTANT_InterfaceMethodref` — these entries are combined into one structure

In the second pass the references are resolved and the runtime structures are created. Any UTF8 strings, `constant_nameandtype` structures or referenced classes are resolved with the

```
voidptr class_getconstant(classinfo *c, u4 pos, u4 ctype);
```

function. This functions checks for type equality and then returns the requested `cpinfos` slot of the specified class.

### 2.2.2 Interface loading

Interface loading is very simple and straightforward. After reading the number of interfaces, for every interface referenced, a `u2` constant pool index is read from the currently loading class or interface. This index is used to resolve the interface class via the `class_getconstant` function from the class' constant pool. This means, interface *loading* is more interface *resolving* than loading. The resolved interfaces are stored in an `classinfo *` array allocated by the CACAO class loader system. The memory pointer of the array is assigned to the `interfaces` field of the `clasinfo` structure.

### 2.2.3 Field loading

The number of fields—`fields_count`—of the class or interface is read as `u2` value from the binary representation of the class or interface (see figure 2.3).

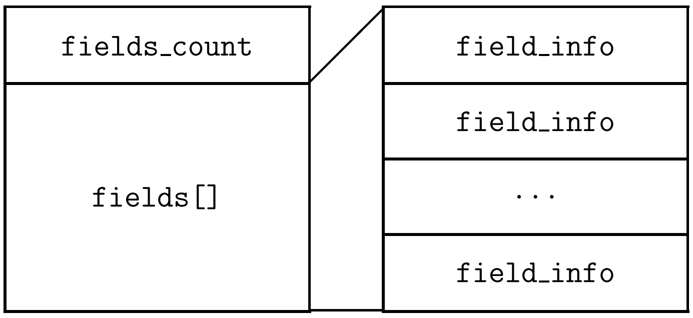

Figure 2.3: Fields area in binary representation of a class or interface

For each field the function

```
static bool field_load(classbuffer *cb, classinfo *c, fieldinfo *f);
```

is called. The `fieldinfo *` argument is a pointer to a `fieldinfo` structure (see figure 2.4) allocated by the CACAO class loader system. The fields' `name` and `descriptor` are resolved from the class constant pool via `class_getconstant`. If the verifier option is turned on, the fields' `flags`, `name` and `descriptor` are checked for validity and can result in throwing a `java.lang.ClassFormatError`.

Each field can have some attributes. The number of attributes is read as `u2` value from the binary representation. If the field has the `ACC_FINAL`

```
struct fieldinfo {        /* field of a class           */
    s4   flags;           /* ACC flags                  */
    s4   type;            /* basic data type            */
    utf *name;            /* name of field              */
    utf *descriptor;      /* JavaVM descriptor string of */
                          /* field                      */
    s4   offset;          /* offset from start of object */
                          /* (instance variables)       */
    imm_union  value;     /* storage for static values   */
                          /* (class variables)          */
    ...
};
```

Figure 2.4: `fieldinfo` structure

bit set in the flags, the `ConstantValue` attribute is available. This is the only attribute processed by `field_load` and can occur only once, otherwise a `java.lang.ClassFormatError` is thrown. The `ConstantValue` entry in the constant pool contains the value for a `final` field. Depending on the fields' type, the proper constant pool entry is resolved and assigned.

### 2.2.4 Method loading

Like for fields, the number of methods of the class or interface—`methods_count`—is read from the binary representation (see figure 2.5) as `u2` value.

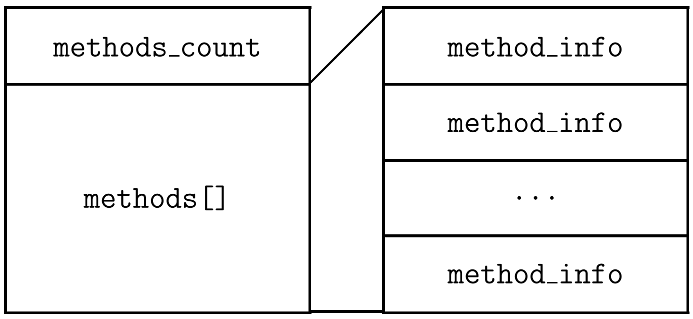

Figure 2.5: Method area in binary representation of a class or interface

For each method found the function

```
static bool method_load(classbuffer *cb, classinfo *c, methodinfo *m);
```

is called. The `methodinfo *` argument is a pointer to a `methodinfo` structure (see figure 2.6) allocated by the CACAO class loading system. The method's `name` and `descriptor` are resolved from the class constant pool via `class_getconstant`. With the verifier turned on, some method checks are carried out. These include `flags`, `name` and `descriptor` checks and argument count check.

The method loading function has to distinguish between a `native` and a "normal" Java method. Depending on the `ACC_NATIVE` flag bit a different stub is created.

For a Java method, a *compiler stub* is created. The purpose of this stub is to call the CACAO JIT compiler with a pointer to the byte code of the Java method as argument to compile the method into machine code. During code generation the pointer to this compiler stub routine is used as temporary method call target, if the method is used in a function call and is not compiled yet. After the target method is compiled, the new entry point of the method is patched into the generated code and the compiler stub is needless, thus it is freed.

If the method is a `native` method, the CACAO class loader system tries to find the native function. If the function was found, a *native stub* is generated. This stub is responsible to manipulate the method's arguments to be suitable for the `native` method called. This includes inserting the *JNI environment* pointer as first argument and, if the `native` method has the `ACC_STATIC` flag set, inserting a pointer to the methods class as second argument. If the `native` method is `static`, the native stub also checks if the method's class is already initialized. If the method's class is not initialized whilst the native stub is generated, a `asm_check_clinit` calling code is emitted.

Each method found in the binary representation can have some attributes. The method loading function processes two of them: `Code` and `Exceptions`.

The `Code` attribute is a *variable-length* attribute which contains the Java Virtual Machine instructions—the byte code—of the Java method. If the method is either `native` or `abstract`, it must not have a `Code` attribute, otherwise it must have exactly one `Code` attribute. Additionally to the byte code, the `Code` attribute contains the exception table and attributes to `Code` attribute itself. One exception table entry contains the `start_pc`, `end_pc` and `handler_pc` of the `try-catch` block, each read as `u2` value, plus a reference to the class of the `catch_type`. Currently there are two attributes of the `Code` attribute defined in the Java Virtual Machine specification: `LineNumberTable` and `LocalVariableTable`. CACAO only processes

```
struct methodinfo {                     /* method structure                      */
    java_objectheader header;           /* we need this in jit's monitorenter    */
    s4          flags;                  /* ACC flags                             */
    utf        *name;                   /* name of method                        */
    utf        *descriptor;             /* JavaVM descriptor string of method    */
    ...
    bool        isleafmethod;           /* does method call subroutines          */

    classinfo  *class;                  /* class, the method belongs to          */
    s4          vftblindex;             /* index of method in virtual function   */
                                        /* table (if it is a virtual method)     */
    s4          maxstack;               /* maximum stack depth of method         */
    s4          maxlocals;              /* maximum number of local variables     */
    s4          jcodelength;            /* length of JavaVM code                 */
    u1         *jcode;                  /* pointer to JavaVM code                */
    ...
    s4          exceptiontablelength;/* exceptiontable length                    */
    exceptiontable *exceptiontable; /* the exceptiontable                        */

    u2          thrownexceptionscount;/* number of exceptions attribute          */
    classinfo **thrownexceptions;       /* checked exceptions a method may throw */

    u2          linenumbercount;        /* number of linenumber attributes       */
    lineinfo   *linenumbers;            /* array of lineinfo items               */
    ...
    u1         *stubroutine;            /* stub for compiling or calling natives */
    ...
};
```

Figure 2.6: `methodinfo` structure

the `LineNumberTable` attribute. A `LineNumberTable` entry consist of the `start_pc` and the `line_number`, which are stored in a `lineinfo` structure.

The linenumber count and the memory pointer of the `lineinfo` structure array are assigned to the `classinfo` fields `linenumbercount` and `linenumbers` respectively.

The `Exceptions` attribute is a *variable-length* attribute and contains the checked exceptions the Java method may throw. The `Exceptions` attribute consist of the count of exceptions, which is stored in the `classinfo` field `thrownexceptionscount`, and the adequate amount of `u2` constant pool index values. The exception classes are resolved from the constant pool and stored in an allocated `classinfo *` array, whose memory pointer is assigned to the `thrownexceptions` field of the `classinfo` structure.

Any attributes which are not processed by the CACAO class loading system are skipped via

```
static bool skipattributebody(classbuffer *cb);
```

which skips one attribute or

```
static bool skipattributes(classbuffer *cb, u4 num);
```

which skips a specified number `num` of attributes. If any problem occurs in the method loading function, a `java.lang.ClassFormatError` with a specific detail message is thrown.

### 2.2.5 Attribute loading

Attribute loading is done via the

```
static bool attribute_load(classbuffer *cb, classinfo *c,
                           u4 num);
```

function from the binary representation of the class or interface (see figure 2.7).

The currently loading class or interface can contain some additional attributes which have not already been loaded. The CACAO system class loader processes two of them: `InnerClasses` and `SourceFile`.

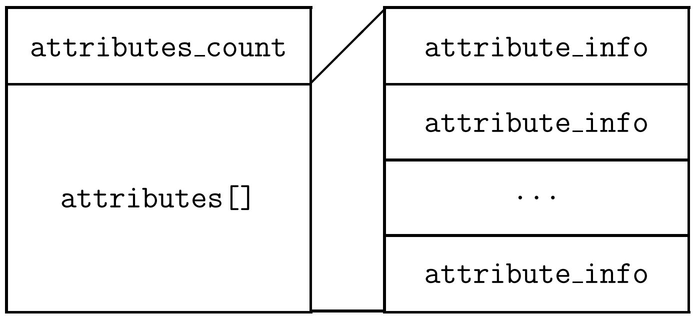

Figure 2.7: Attribute area in binary representation of a class or interface

The `InnerClass` attribute is a *variable-length* attribute in the `attributes` table of the binary representation of the class or interface. A `InnerClass` entry contains the `inner_class` constant pool index itself, the `outer_class` index, the `name` index of the inner class' name and the inner class' `flags` bit mask. All these values are read in `u2` chunks.

The constant pool indexes are used with the

```
voidptr innerclass_getconstant(classinfo *c, u4 pos, u4 ctype);
```

function call to resolve the classes or UTF8 strings. After resolving is done, all values are stored in an `innerclassinfo` structure.

The other attribute, `SourceFile`, is just one `u2` constant pool index value to get the UTF8 string reference of the class' `SourceFile` name. The string pointer is assigned to the `sourcefile` field of the `classinfo` structure.

Both attributes must occur only once. Other attributes than these two are skipped with the earlier mentioned `skipattributebody` function.

After the attribute loading is done and no error occured, the `class_load_intern` function returns the `classinfo` pointer to signal that no problem occured. If `NULL` is returned an error or exception was raised.

## 2.3 Eager - lazy class loading

A Java Virtual Machine can implement two different algorithms for the system class loader to load classes or interfaces: *eager class loading* and *lazy class loading*.

### 2.3.1 Eager class loading

The Java Virtual Machine initially creates, loads and links the class of the main program with the system class loader. The creation of the class is done via the `class_new` function call (see section 2.2). In this function, with *eager loading* enabled, firstly the currently created class or interface is loaded with `class_load`. CACAO uses the eager class loading algorithm with the command line switch `-eager`. As described in the "Constant pool loading" section (see 2.2.1), the binary representation of a class or interface contains references to other classes or interfaces. With eager class loading enabled, referenced classes or interfaces are loaded immediately.

If a class reference is found in the second pass of the constant pool loading process, the class is created in the class hashtable with `class_new_intern`. CACAO uses the intern function here because the normal `class_new` function, which is a wrapper function, instantly tries to *link* all referenced classes when eager class loading is enabled. This must not happen until all classes or interfaces referenced are loaded, otherwise the Java Virtual Machine gets into an indefinite state.

After the `classinfo` of the class referenced is created, the class or interface is *loaded* via the `class_load` function (described in more detail in section 2.2). When the class loading function returns, the current referenced class or interface is added to a list called `unlinkedclasses`, which contains all loaded but unlinked classes referenced by the currently loaded class or interface. This list is processed in the `class_new` function of the currently created class or interface after `class_load` returns. For each entry in the `unlinkedclasses` list, `class_link` is called which finally *links* the class (described in more detail in section 2.4) and then the class entry is removed from the list. When all referenced classes or interfaces are linked, the currently created class or interface is linked and the `class_new` functions returns.

### 2.3.2 Lazy class loading

Usually it takes much more time for a Java Virtual Machine to start a program with *eager class loading* than with *lazy class loading*. With *eager class loading*, a typical `HelloWorld` program needs 513 class loads with the current GNU Classpath CACAO is using. When using *lazy class loading*, CACAO only needs 121 class loads for the same `HelloWorld` program. This means with *lazy class loading* CACAO needs to load more than four times less class files. Furthermore CACAO does also *lazy class linking*, which saves much more run-time here.

CACAO's *lazy class loading* implementation does not completely follow the JVM specification. A Java Virtual Machine which implements *lazy class loading* should load and link requested classes or interfaces at runtime. But CACAO does class loading and linking at parse time, because of some problems not resolved yet. That means, if a Java Virtual Machine instruction is parsed which uses any class or interface references, like `JAVA_PUTSTATIC`, `JAVA_GETFIELD` or any `JAVA_INVOKE*` instructions, the referenced class or interface is loaded and linked immediately during the parse pass of currently compiled method. This introduces some incompatibilities with other Java Virtual Machines like Sun's JVM, IBM's JVM or Kaffe.

Given a code snippet like this

```
void sub(boolean b) {
    if (b) {
        new A();
    }
    System.out.println("foobar");
}
```

If the function is called with `b` equal `false` and the class file `A.class` does not exist, a Java Virtual Machine should execute the code without any problems, print `foobar` and exit the Java Virtual Machine with exit code 0. Due to the fact that CACAO does class loading and linking at parse time, the CACAO Virtual Machine throws an `java.lang.NoClassDefFoundError: A` exception which is not caught and thus discontinues the execution without printing `foobar` and exits.

The CACAO development team has not yet a solution for this problem. It's not trivial to move the loading and linking process from the compilation phase into runtime, especially CACAO was initially designed for *eager class loading* and *lazy class loading* was implemented at a later time to optimize class loading and to get a little closer to the JVM specification. *Lazy class loading* at runtime is one of the most important features to be implemented in the future. It is essential to make CACAO a standard compliant Java Virtual Machine.

## 2.4 Linking

Linking is the process of preparing a previously loaded class or interface to be used in the Java Virtual Machine's runtime environment. The function which performs the linking in CACAO is

```
classinfo *class_link(classinfo *c);
```

This function, as for class loading, is just a wrapper function to the main linking function

```
static classinfo *class_link_intern(classinfo *c);
```

This function should not be called directly and is thus declared as `static`. The purposes of the wrapper function are

- enter a monitor on the `classinfo` structure, so that only one thread can link the same class or interface at the same time
- check if the class or interface is `linked`, if it is `true`, leave the monitor and return immediately
- measure linking time if requested
- check if the intern linking function has thrown an error or an exception and reset the `linked` field of the `classinfo` structure to `false`
- leave the monitor

The `class_link` function, like the `class_load` function, is implemented to be *reentrant*. This must be the case for the linking algorithm implemented in CACAO. Furthermore this means that serveral threads can link different classes or interfaces at the same time on multiprocessor machines.

The first step in the `class_link_intern` function is to set the `linked` field of the currently linked `classinfo` structure to `true`. This is essential, that the linker does not try to link a class or interface again, while it's already in the linking process. Such a case can occur because the linker also processes the class' direct superclass and direct superinterfaces.

In CACAO's linker the direct superinterfaces are processed first. For each interface in the `interfaces` field of the `classinfo` structure is checked if there occured an `java.lang.ClassCircularityError`, which happens when the currently linked class or interface is equal the interface which should be processed. Otherwise the interface is loaded and linked if not already done. After the interface is loaded successfully, the interface flags are checked for the `ACC_INTERFACE` bit. If this is not the case, a `java.lang.IncompatibleClassChangeError` is thrown and `class_link_intern` returns.

Then the direct superclass is handled. If the direct superclass is equal `NULL`, we have the special case of linking `java.lang.Object`. There are only set some `classinfo` fields to special values for `java.lang.Object` like

```
c->index = 0;
c->instancesize = sizeof(java_objectheader);
vftbllength = 0;
c->finalizer = NULL;
```

If the direct superclass is non-`NULL`, CACAO firstly detects class circularity as for interfaces. If no `java.lang.ClassCircularityError` was thrown, the superclass is loaded and linked if not already done before. Then some flag bits of the superclass are checked: `ACC_INTERFACE` and `ACC_FINAL`. If one of these bits is set an error is thrown.

If the currently linked class is an array, CACAO calls a special array linking function

```
static arraydescriptor *class_link_array(classinfo *c);
```

This function firstly checks if the passed `classinfo` is an *array of arrays* or an *array of objects*. In both cases the component type is created in the class hashtable via `class_new` and then loaded and linked if not already done. If none is the case, the passed array is a *primitive type array*. No matter of which type the array is, an `arraydescriptor` structure is allocated and filled with the appropriate values of the given array type.

After the `class_link_array` function call, the class `index` is calculated. For interfaces—classes with `ACC_INTERFACE` flag bit set—the class' `index` is the global `interfaceindex` plus one. Any other classes get the `index` of the superclass plus one.

Some `classinfo` fields are inherited from the superclass like `instancesize`, `vftbllength` and the `finalizer` function. All these values are temporary ones and can be overwritten at a later time.

The next step in `class_load_intern` is to compute the *virtual function table length*. For each method in `classinfo`'s `methods` field which has not the `ACC_STATIC` flag bit set, thus is an instance method, the direct superclasses up to `java.lang.Object` are checked with

```
static bool method_canoverwrite(methodinfo *m, methodinfo *old);
```

if the current method can overwrite the superclass method, if there exists one. If the superclass method has the `ACC_FINAL` flag bit set, the CACAO class linker throws a `java.lang.VerifyError`.

```
struct vftbl {
    methodptr   *interfacetable[1];    /* interface table (access via macro)  */

    classinfo   *class;                /* class, the vtbl belongs to          */

    arraydescriptor *arraydesc;        /* for array classes, otherwise NULL   */

    s4           vftbllength;          /* virtual function table length       */
    s4           interfacetablelength; /* interface table length              */

    s4           baseval;              /* base for runtime type check         */
                                       /* (-index for interfaces)             */
    s4           diffval;              /* high - base for runtime type check  */

    s4          *interfacevftbllength; /* length of interface vftbls          */

    methodptr    table[1];             /* class vftbl                         */
};
```

Figure 2.8: `vftbl` structure

After processing the *virtual function table length*, the CACAO linker computes the *interface table length*. For the current class' and every superclass' interfaces, the function

```
static s4 class_highestinterface(classinfo *c);
```

is called. This function computes the highest interface `index` of the passed interface and returns the value. This is done by recursively calling `class_highestinterface` with each interface from the `interfaces` array of the passed interface as argument. The highest `index` value found is the *interface table length* of the currently linking class or interface.

Now that the linker has completely computed the size of the *virtual function table*, the memory can be allocated, casted to an `vftbl` structure (see figure 2.8) and filled with the previously calculated values.

Afterwards the fields of the currently linked class or interface are processed. The CACAO linker computes the instance size of the class or interface and the offset of each field inside. For each field in the `classinfo` field `fields` which is non-`static`, the type-size is resolved via the `desc_typesize` function call. Then a new `instancesize` is calculated with

```
c->instancesize = ALIGN(c->instancesize, dsize);
```

which does memory alignment suitable for the next field. This newly computed `instancesize` is the `offset` of the currently processed field. The type-size is then added to get the real `instancesize`.

The next step of the CACAO linker is to initialize two virtual function table fields, namely `interfacevftbllength` and `interfacetable`. After the initialization is done, the interfaces of the currently linked class and all it's superclasses, up to `java.lang.Object`, are processed via the

```
static void class_addinterface(classinfo *c, classinfo *ic);
```

function call. This function adds the methods of the passed interface to the *virtual function table* of the passed class or interface. If the method count of the passed interface is zero, the function adds a method fake entry, which is needed for subtype tests.

For each method of the passed interface, the methods of the passed target class or interface and all superclass methods, up to `java.lang.Object`, are checked if they can overwrite the interface method via `method_canoverwrite`.

The `class_addinterface` function is also called recursively for all interfaces the interface passed implements.

After the interfaces were added and the currently linked class or interface is not `java.lang.Object`, the CACAO linker tries to find a function which name and descriptor matches `finalize()V`. If an appropriate function was found and the function is non-`static`, it is assigned to the `finalizer` field of the `classinfo` structure. CACAO does not assign the `finalize()V` function to `java.lang.Object`, because this function is inherited to all subclasses which do not explicitly implement a `finalize()V` method. This would mean, for each instantiated object, which is marked for garbage collection in the Java Virtual Machine, an empty function would be called from the garbage collector when a garbage collection takes place. This would add an needless overhead to a garbage collection run.

The final task of the linker is to compute the `baseval` and `diffval` values from the subclasses of the currently linked class or interface. These values are used for *runtime type checking*.

After the `baseval` and `diffval` values are newly calculated for all classes and interfaces in the Java Virtual Machine, the internal linker function `class_link_intern` returns the currently linking `classinfo` structure pointer, to indicate that the linker function did not raise an error or exception.

## 2.5 Initialization

A class or interface can have a `static` initialization function called *static class initializer*. The function has the name and signature `<clinit>()V`. This function must be invoked before a `static` function of the class is called or a `static` field is accessed via `ICMD_PUTSTATIC` or `ICMD_GETSTATIC`. In CACAO

```
classinfo *class_init(classinfo *c);
```

is responsible for the invocation of the *static class initializer*. It is, like for class loading and class linking, just a wrapper function to the main initializing function

```
static classinfo *class_init_intern(classinfo *c);
```

The wrapper function has the following purposes:

- enter a monitor on the `classinfo` structure, so that only one thread can initialize the same class or interface at the same time
- check if the class or interface is `initialized` or `initializing`, if one is `true`, leave the monitor and return
- tag the class or interface as `initializing`
- call the internal initialization function `class_init_intern`
- if the internal initialization function returns non-`NULL`, the class or interface is tagged as `initialized`
- reset the `initializing` flag
- leave the monitor

The intern initializing function should not be called directly, because of race conditions of concurrent threads. Two or more different threads could access a `static` field or call a `static` function of an uninitialized class at almost the same time. This means that each single thread would invoke the static class initializer and this would lead into some problems.

The CACAO initializer needs to tag the class or interface as currently initializing. This is done by setting the `initializing` field of the `classinfo` structure to `true`. CACAO needs this field in addition to the `initialized` field for two reasons:

- Another concurrently running thread can access a `static` field of the currently initializing class or interface. If the class or interface of the `static` field was not initialized during code generation, some special code was emitted for the `ICMD_PUTSTATIC` and `ICMD_GETSTATIC` intermediate commands. This special code is a call to an architecture dependent assembler function named `asm_check_clinit`. Since this function is speed optimized for the case that the target class is already initialized, it only checks for the `initialized` field and does not take care of any monitor that may have been entered. If the `initialized` flag is `false`, the assembler function calls the `class_init` function where it probably stops at the monitor enter. Due to this fact, the thread which does the initialization can not set the `initialized` flag to `true` when the initialization starts, otherwise potential concurrently running threads would continue their execution although the static class initializer has not finished yet.

- The thread which is currently `initializing` the class or interface can pass the monitor which has been entered and thus needs to know if this class or interface is currently initialized.

Firstly `class_init_intern` checks if the passed class or interface is loaded and linked. If not, the particular action is taken. This is just a safety measure, because—CACAO internally—each class or interface should have already been loaded and linked before `class_init` is called.

Then the superclass, if any specified, is checked if it is already initialized. If not, the initialization is done immediately. The same check is performed for each interface in the `interfaces` array of the `classinfo` structure of the current class or interface.

After the superclass and all interfaces are initialized, CACAO tries to find the static class initializer function. If no static class initializer method is found in the current class or interface, the `class_link_intern` functions returns immediately without an error. If a static class initializer method is found, it's called with the architecture dependent assembler function

```
java_objectheader *asm_calljavafunction(methodinfo *m, void *arg1, void *arg2,
                                         void *arg3, void *arg4);
```

Exception handling of an exception thrown in an static class initializer is a bit different than usual. It depends on the type of exception. If the exception thrown is an instance of `java.lang.Error`, the `class_init_intern` function just returns `NULL`. If the exception thrown is an instance of `java.lang.Exception`, the exception is wrapped into a `java.lang.ExceptionInInitializerError`. This is done via the `new_exception_throwable` function call. The newly generated error is set as exception thrown and the `class_init_intern` returns `NULL`.

If no exception occurred in the static class initializer the internal initializing function returns the current `classinfo` structure pointer to indicate that the initialization was successful.

# Chapter 3

# The Just-In-Time Compiler

## 3.1 Introduction

A Java Virtual Machine can implement four different approaches of executing Java byte code:

- Interpreter
- Ahead-Of-Time Compiler
- Just-In-Time Compiler
- Mixed Mode

An *Interpreter* interprets every single virtual machine instruction in the language the Java Virtual Machine is written in, mainly C. Due to this fact an interpreter based Java Virtual Machine is easily portable, but the execution speed is very slow, up to ten times slower than a current Just-In-Time Compilers or similar code written in C.

An *Ahead-Of-Time Compiler* compiles every Java method of a class when the class is loaded. This reduces the compiler overhead since the compilation of the class methods is done in one step and at runtime the method called can be executed immediately. The drawback of this approach is the fact that every single method is compiled, even if it's not needed. This can use a serious amount of memory and time since the java libraries contain a lot of methods.

A *Just-In-Time Compiler* is the solution to the memory and compilation time problem of the Ahead-Of-Time compiler. A Just-In-Time compiler only compiles a method when it is called the first time. The drawback of this approach is the deferred execution of the called method since it must be compiled before.

The *Mixed Mode* is mostly a mixture of an Interpreter and a Just-In-Time Compiler. Normally the code is interpreted, but code parts which are frequently executed are compiled to native machine code with an Just-In-Time Compiler. This technique is used by Sun's and IBM's JVM.

CACAO implements a *compile-only* approach with a *Just-In-Time Compiler* and has no interpreter. The main target of CACAO was to build a fast executing Java Virtual Machine with short compilation times. Thus the CACAO development team decided to only implement a fast compiling Just-In-Time Compiler. So every single Java method executed is compiled to native machine code.

The following sections decribe some basics of the Java Virtual Machine, byte code to machine code compilation and how the CACAO Just-In-Time Compiler works in detail.

## 3.2 The Java Virtual Machine

The JavaVM is a typed stack architecture [LY99]. There are different instructions for integer, long integer, floating point and address types. Byte and character types have only special memory access instructions and are treated as integers for arithmetic operations. The main instruction set consists of arithmetic/logical and load/store/constant instructions. There are special instructions for array access and for accessing the fields of objects (memory access), for method invocation and for type checking. A JavaVM has to check the program for type correctness and executes only correct programs. The following examples show some important JavaVM instructions and how a Java program is represented by these instructions.

```
iload  n    ; load contents of local variable n
istore n    ; store stack top in local variable n
imul        ; product of 2 topmost stack elements
isub        ; difference of 2 topmost stack elements
```

The Java assignment statement `a = b - c * d` is translated into

```
iload b     ; load contents of variable b
iload c     ; load contents of variable c
iload d     ; load contents of variable d
imul        ; compute c * d
isub        ; compute b - (c * d)
istore a    ; store stack top in variable a
```

Accessing the fields of objects is handled by the instructions `getfield` and `putfield`. The instruction `getfield` expects an object reference on the stack and has an index into the constant pool as an operand. The index into the constant pool must be a reference to a pair containing the class name and a field name. The types of the classes referenced by the constant pool index and by the object reference must be compatible, a fact which is usually checked statically at load time. The object reference has to be different from the `null` pointer, a fact which must usually be checked at run time.

Array accesses are handled by the `aload` and `astore` instructions. Separate versions of these instructions exist for each of the basic types (`byte`, `int`, `float`, `ref`, etc.). The `aload` instruction expects a reference to an array and an index (of type `int`) on the stack. The array reference must not be the `null` pointer. The index must be greater than or equal to zero and less than the array length.

There are special commands for method invocation. Each method has its own virtual stack and an area for local variables. After the method invocation, the stack of the caller is empty and the arguments are copied into the first local variables of the called method. After execution of a `return` instruction, the called method returns to its caller. If the called method is a function, it pops the return value from its own stack and pushes it onto the stack of the caller.

The `instanceof` and `checkcast` instructions are used for subtype testing. Both expect a reference to an object on the stack and have an index into the constant pool as operand. The index must reference a class, array or interface type. The two instructions differ in their result and in their behavior if the object reference is `null`.

## 3.3 Translation of stack code to register code

The architecture of a RISC—*Reduced Instruction Set Computer*—or a CISC—*Complex Instruction Set Computer*—processor is completely different from the stack architecture of the Java Virtual Machine.

RISC processors have large sets of registers. The Alpha architecture has 32 integer registers and 32 floating point registers which are both 64-bits wide. They execute arithmetic and logic operations only on values which are held in registers. RISC machines are a load-store architecture, which means load and store instructions are provided to move data between memory and registers. Local variables of methods usually reside in registers and are saved to memory only during a method call or if there are too few registers. The use of registers for parameter passing is defined by the particular calling conventions.

CISC processors have a relatively small set of registers, like the IA32 architecture with 8 integer general purpose registers or the AMD64 architecture with 16 integer general purpose registers. But, as the name implies, CISC processors have a large and complex machine instruction set. Unlike the load-store architecture of RISC machines, CISC instructions can operate on operands residing in registers and in memory locations. Local variables of methods should reside in registers, but due to the limited number of registers this case is very rare and most local variables are stored on the stack. Detailed information of the IA32 and AMD64 architecture can be found in section 3.6 and section 3.7 respectively.

### 3.3.1 Machine code translation examples

The previous expression example `a = b - c * d` would be translated by an optimizing C compiler to the following two Alpha instructions (the variables `a`, `b`, `c` and `d` reside in registers):

```
MULL c,d,tmp0     ; tmp0 = c * d
SUBL b,tmp0,a     ; a = b - tmp0
```

If JavaVM code is translated to machine code, the stack is eliminated and the stack slots are represented by temporary variables usually residing in registers. A naive translation of the previous example would result in the following Alpha instructions:

```
MOVE b,t0         ; iload b
MOVE c,t1         ; iload c
MOVE d,t2         ; iload d
MULL t1,t2,t1     ; imul
SUBL t0,t1,t0     ; isub
MOVE t0,a         ; istore a
```

The problems of translating JavaVM code to machine code are primarily the elimination of the unnecessary copy instructions and finding an efficient register allocation algorithm. A common but expensive technique is to do the naive translation and use an additional pass for copy elimination and coalescing.

## 3.4 The translation algorithm

The new translation algorithm can get by with three passes. The first pass determines basic blocks and builds a representation of the JavaVM instructions which is faster to decode. The second pass analyses the stack and generates a static stack structure. During stack analysis variable dependencies are tracked and register requirements are computed. In the final pass register allocation of temporary registers is combined with machine code generation.

The new compiler computes the exact number of objects needed or computes an upper bound and allocates the memory for the necessary temporary data structures in three big blocks: the basic block array, the instruction array and the stack array..

### 3.4.1 Basic block determination

The first pass scans the JavaVM instructions, determines the basic blocks and generates an array of instructions which has a fixed size and is easier to decode in the following passes. Each instruction contains the opcode, two operands and a pointer to the static stack structure after the instruction (see next sections). The different opcodes of JavaVM instructions which fold operands into the opcode are represented by just one opcode in the instruction array.

### 3.4.2 Basic block interfacing convention

Basic blocks have a fixed interface at basic block boundaries. Every stack slot at a basic block boundary is assigned a fixed interface register. The stack analysis pass determines the type of the register and if it has to be saved across method invocations. To enlarge the size of basic blocks method invocations do not end basic blocks. To guide the compiler design some static analysis on a large application written in Java was done: the `javac` compiler

| stack depth | 0 | 1 | 2 | 3 | 4 | 5 | 6 | >6 |
|---|---|---|---|---|---|---|---|---|
| occurrences | 7930 | 258 | 136 | 112 | 36 | 8 | 3 | 0 |

Table 3.1: distribution of stack depth at block boundary

and the libraries it uses. Table 3.1 shows that in more than 93% of the cases the stack is empty at basic block boundaries and that the maximal stack depth is 6. Using this data it becomes quite clear that some join handling would not improve the quality of the machine code.

### 3.4.3 Copy elimination

To eliminate unnecessary copies, the loading of values is delayed until the instruction is reached which consumes the value. To compute the information the run time stack is simulated at compile time. Instead of values the compile time stack contains the type of the value, if a local variable was loaded to a stack location and similar information. Adl-Tabatabai [ATCL$^{+}$98] used a dynamic stack which is changed at every instruction. A dynamic stack only gives the possibility to move information from earlier instructions to later instructions. CACAO uses a static stack structure which enables information flow in both directions.

Figure 3.1 shows the instruction and stack representation. An instruction has a reference to the stack before the instruction and the stack after the instruction. The stack is represented as a linked list. The two stacks can be seen as the source and destination operands of an instruction. In the implementation only the destination stack is stored, the source stack is the destination of stack of the previous instruction.

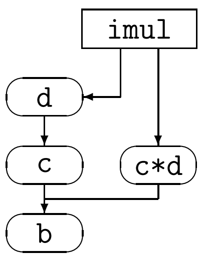

Figure 3.1: instruction and stack representation

This representation can easily be used for copy elimination. Each stack element not only contains the type of the stack slot but also the local variable number of which it is a copy, the argument number if it is an argument, the interface register number if it is an interface. Load (push the content of a variable onto the stack) and store instructions do no generate a copy machine instruction if the stack slot contains the same local variable. Generated machine instructions for arithmetic operations directly use the local variables as their operands.

There are some pitfalls with this scheme. Given the example of figure 3.2.

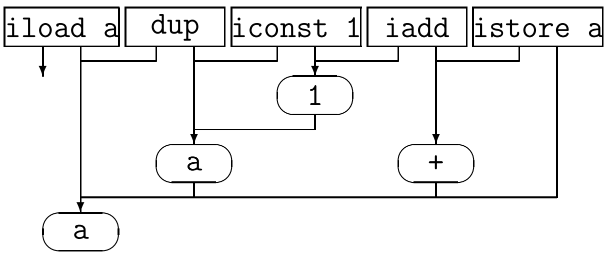

Figure 3.2: anti dependence

The stack bottom contains the local variable `a`. The instruction `istore a` will write a new value for `a` and will make a later use of this variable invalid. To avoid this a copy of the local variable to a stack variable is necessary. An important decision is at which position the copy instruction should be inserted. Since there is a high number of `dup` instructions in Java programs (around 4%) and it is possible that a local variable resides in memory, the copy should be done with the `load` instruction. Since the stack is represented as a linked list only the destination stack has to be checked for occurrences of the offending variable and these occurrences are replaced by a stack variable.

To answer the question of how often this could happen and how expensive the stack search is, again the `javac` compiler was analysed. In more than 98% of the cases the stack is empty (see table 3.2). In only 0.2% of the cases the stack depth is higher than 1 and the biggest stack depth is 3.

| stack depth | 0 | 1 | 2 | 3 | >3 |
|---|---|---|---|---|---|
| occurrences | 2167 | 31 | 1 | 3 | 0 |

Table 3.2: distribution of `store` stack depth

To avoid copy instructions when executing a `store` it is necessary to connect the creation of a value with the store which consumes it. In that case a `store` not only can conflict with copies of a local variable which result from `load` instructions before the creator of the value, but also with `load` and `store` instructions which exist between the creation of value and the `store`. In figure 3.3 the `iload a` instruction conflicts with the `istore a` instruction.

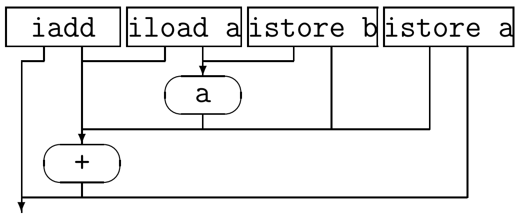

Figure 3.3: anti dependence

The anti dependences are detected by checking the stack locations of the previous instructions for conflicts. Since the stack locations are allocated as one big array just the stack elements which have a higher index than the current stack element have to be checked. Table 3.3 gives the distribution of the distance between the creation of the value and the corresponding store. In 86% of the cases the distance is one.

| chain length | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | >9 |
|---|---|---|---|---|---|---|---|---|---|---|
| occurrences | 1892 | 62 | 23 | 62 | 30 | 11 | 41 | 9 | 7 | 65 |

Table 3.3: distribution of creator-store distances

The output dependences are checked by storing the instruction number of the last store in each local variable. If a store conflicts due to dependences the creator places the value in a stack register. Additional dependences arise because of exceptions. The exception mechanism in Java is precise. Therefore `store` instructions are not allowed to be executed before an exception raising instruction. This is checked easily be remembering the last instruction which could raise an exception. In methods which contain no exception handler this conflict can be safely ignored because no exception handler can have access to these variables.

### 3.4.4 Register allocator

he current register allocator of CACAO is a very simple, straightforward allocator. It simply assigns free registers with a *first-come-first-serve* based algorithm. This is mostly suitable for RISC architectures with large register sets like Alpha or MIPS with 32 integer registers and 32 floating-point registers. For these architectures the current register allocator was designed for.

Basically the allocation passes of the register allocator are:

- interface register allocation
- scratch register allocation
- local register allocation

The `javac` compiler also supports this simple *first-come-first-serve* approach CACAO uses and does a coloring of the local variables and assigns the same number to variables which are not active at the same time. The stack variables have implicitly encoded their live ranges. When a value is pushed, the live range start. When a value is popped, the live range ends.

Complications arise only with stack manipulation instructions like `dup` and `swap`. We flag therefore the first creation of a stack variable and mark a duplicated one as a copy. The register used for this variable can be reused only after the last copy is popped.

During stack analysis stack variables are marked which have to survive a method invocation. These stack variables and local variables are assigned to callee saved registers. If there are not enough registers available, these variables are allocated in memory.

Efficient implementation of method invocation is crucial to the performance of Java. Therefore, we preallocate the argument registers and the return value in a similar way as we handle store instructions. Input arguments (in Java input arguments are the first variables) for leaf procedures (and input arguments for processors with register windows) are preassigned, too.

Since CACAO has now also been ported to CISC architectures like IA32 and AMD64, the *first-come-first-serve* register allocator has hit it's limits. The results produced for an architecture with 8 integer general purpose registers like IA32 or 16 integer general purpose registers like AMD64, is far from perfect. Further details to register allocation of these architectures can be found in section 3.6.5 and section 3.7.5 respectively.

The CACAO development team is currently working on a new register allocator based on a *linear scan* algorithm. This allocator should produce much better results on CISC machines than the current register allocator.

### 3.4.5 Instruction combining

Together with stack analysis we combine constant loading instructions with selected instructions which are following immediately. In the class of combinable instructions are add, subtract, multiply and divide instructions, logical and shift instructions, compare/branch and array store instructions.

These combined immediate instructions are:

- `ICMD_IADDCONST`, `ICMD_ISUBCONST`, `ICMD_IMULCONST`, `ICMD_IDIVPOW2`, `ICMD_IREMPOW2`
- `ICMD_LADDCONST`, `ICMD_LSUBCONST`, `ICMD_LMULCONST`, `ICMD_LDIVPOW2`, `ICMD_LREMPOW2`
- `ICMD_IANDCONST`, `ICMD_IORCONST`, `ICMD_IXORCONST`
- `ICMD_LANDCONST`, `ICMD_LORCONST`, `ICMD_LXORCONST`
- `ICMD_ISHLCONST`, `ICMD_ISHRCONST`, `ICMD_IUSHRCONST`
- `ICMD_LSHLCONST`, `ICMD_LSHRCONST`, `ICMD_LUSHRCONST`
- `ICMD_IFxx`
- `ICMD_IF_Lxx`
- `ICMD_xASTORECONST`

During code generation the constant is checked if it lies in the range for immediate operands of the target architecture and appropriate code is generated.

Arithmetic and logical instructions are processed straightforward. The intermediate command opcode of the current instruction is changed and the immediate value from the previous instruction is stored in the current instruction. The register pressure is always reduced by one register by this optimization.

`ICMD_IDIV` and `ICMD_IREM` divisions by a constant which is a power of two are handled in a special way. They are converted into `ICMD_IDIVPOW2` and `ICMD_IREMPOW2` respectively. For `ICMD_IDIVPOW2` an immediate value is assigned which represents the left shift amount of `0x1` to get the divisor value. In the code generation pass a very fast shift-based machine code can be generated for this instruction. For `ICMD_IREMPOW2` the intermediate value gets one subtracted. The generated machine code consists of logical `and`'s, `neg`'s and a conditional jump. For both instructions the generated machine code is much fast than an integer division. `ICMD_LDIV` and `ICMD_LREM` intermediate commands are handled respectively.

`ICMD_IxSHx` instructions by a constant value are converted to `ICMD_IxSHxCONST` instructions. Nearly every architecture has machine shift instructions by a constant value. This optimization always reduces the register pressure by one register. `ICMD_LxSHx` intermediate commands are converted to `ICMD_LxSHxCONST` commands respectively.

`ICMD_IF_ICMPxx` intermediate commands are converted to `ICMD_IFxx` commands. This commands compare the source operand directly with an immediate value if possible. The generated machine code depends on the architecture. On the IA32 or AMD64 architecture the immediate value can always be inlined. On RISC architectures the immediate value range is limited, like the Alpha architecture where the immediate value may be between 0 and 255. On architectures which support conditional branches on a source register, like Alpha or MIPS, the compare with 0 is optimized to a single instruction. This optimization can reduce the register pressure by one register. `ICMD_IF_Lxx` intermediate commands are handled respectively.

The `ICMD_xASTORE` optimization was actually implemented for the IA32 and AMD64 architecture. These architectures can handle inline immediate values up to their address pointer size, this means 32-bit for IA32 and 64-bit for AMD64 respectively. For RISC architectures which have a `REG_ZERO`—a register which always contains the values zero—this array store optimization can be used only for zero values. Address array stores—`ICMD_AASTORE`—can only be optimized in the `null` pointer case because of the dynamic type check. In this case the optimization not only reduces the register pressure by one register, but the dynamic type check subroutine call can be eliminated.

### 3.4.6 Complexity of the algorithm

The complexity of the algorithm is mostly linear with respect to the number of instructions and the number of local variables plus the number of stack slots. There are only a small number of spots where it is not linear.

- At the begin of a basic block the stack has to be copied to separate the stacks of different basic blocks. Table 3.1 shows that the stack at the boundary of a basic block is in most cases zero. Therefore, this copying does not influence the linear performance of the algorithm.

- A store has to check for a later use of the same variable. Table 3.2 shows that this is not a problem, too.
- A store additionally has to check for the previous use of the same variable between creation of the value and the store. The distances between the creation and the use are small (in most case only 1) as shown by table 3.3.

Compiling `javac` 29% of the compile time are spent in parsing and basic block determination, 18% are spent in stack analysis, 16% are spent in register allocation and 37% are spent in machine code generation.

### 3.4.7 Example

Figure 3.4 shows the intermediate representation and stack information as produced by the compiler for debugging purposes. The `Local Table` gives the types and register assignment for the local variables. The Java compiler reuses the same local variable slot for different local variables if there life ranges do not overlap. In this example the variable slot 3 is even used for local variables of different types (integer and address). The JIT-compiler assigned the saved register 12 to this variable.

One interface register is used in this example entering the basic block with label `L004`. At the entry of the basic block the interface register has to be copied to the argument register `A00`. This is one of the rare cases where a more sophisticated coalescing algorithm could have allocated an argument register for the interface.

At instruction 2 and 3 you can see the combining of a constant with an arithmetic instruction. Since the instructions are allocated in an array the empty slot has to be filled with a `NOP` instruction. The `ADDCONSTANT` instruction already has the local variable `L02` as destination, an information which comes from the later `ISTORE` at number 4. Similarly the `INVOKESTATIC` at number 31 has marked all its operands as arguments. In this example all copy (beside the one to the interface register) have been eliminated.

## 3.5 Compiling a Java method

The CACAO JIT compiler is invoked via the

```
methodptr jit_compile(methodinfo *m);
```

function call. This function is just a wrapper function to the internal compiler function

```
static methodptr jit_compile_intern(methodinfo *m);
```

The argument of the compiler function is a pointer to a `methodinfo` structure (see figure 2.6) allocated by the system class loader. This function should not be called directly and thus is declared `static` because the wrapper function has to ensure some conditions:

- enter a monitor on the `methodinfo` structure to make sure that only one thread can compile the same Java method at the same time
- check if the method already has a `entrypoint`, if so the monitor is left and the entrypoint is returned
- measure the compiling time if requested
- call the internal compiler function
- leave the monitor and return the functions' `entrypoint`

The internal compiler function `jit_compile_intern` does the actual compilation of the Java method. It calls the different passes of the JIT compiler.

If the passed Java method does not have a *Code Attribute* (see 2.2.4) a `methodptr` to a `do_nothing_function` is returned.

If the method has the `ACC_STATIC` flag bit set and the methods' class is not yet initialized, `class_init` is called with the methods' class as argument

Then the compiler passes are called:

1. `reg_init`: initializes the register allocator
    - allocates the `registerdata` structure
    - calculate the number of callee saved, temporary and argument registers
2. `reg_setup`: sets up the register allocator data which is changed in every compiler run

3. `codegen_setup`: initializes the code generator
   - allocates the `codegendata` structure
   - allocate code and data memory
4. `parse`: parse pass
   - parse the Java Virtual Machine instructions and convert them into CACAO intermediate commands
   - determine basic blocks
5. `analyse_stack`: analyse stack pass
6. `regalloc`: register allocation pass
7. `codegen`: code generation pass
8. `reg_close`: release all allocated register allocator memory
9. `codegen_close`: release all allocated code generator memory

After all compiler passes were run and no exception or error occured, the `entrypoint` of the compiled method is returned.

The CACAO JIT compiler is designed to be reentrant. This design decision was taken to easily support exception throwing during one of the compiler passes and to support concurrent compilation in different threads running. Concurrent compilation can speed up startup and run time especially on multi processor machines.

```
java.io.ByteArrayOutputStream.write (int)void

Local Table:
     0:               (adr) S15
     1:    (int) S14
     2:    (int) S13
     3:    (int) S12   (adr) S12

Interface Table:
     0:    (int) T24

[                 L00]       0  ALOAD          0
[                 T23]       1  GETFIELD       16
[                 L02]       2  IADDCONST      1
[                 L02]       3  NOP
[                    ]       4  ISTORE         2
[                 L02]       5  ILOAD          2
[             L00 L02]       6  ALOAD          0
[             T23 L02]       7  GETFIELD       8
[             T23 L02]       8  ARRAYLENGTH
[                    ]       9  IF_ICMPLE      L005
                                ...............

[                    ]      18  IF_ICMPLT      L003
[                    ] L002:
[                 I00]      19  ILOAD          3
[                 I00]      20  GOTO           L004
[                    ] L003:
[                 I00]      21  ILOAD          2
[                 A00] L004:
[                 L03]      22  BUILTIN1       newarray_byte
[                    ]      23  ASTORE         3
[                 L00]      24  ALOAD          0
[                 A00]      25  GETFIELD       8
[             A01 A00]      26  ICONST         0
[         A02 A01 A00]      27  ALOAD          3
[     A03 A02 A01 A00]      28  ICONST         0
[ L00 A03 A02 A01 A00]      29  ALOAD          0
[ A04 A03 A02 A01 A00]      30  GETFIELD       16
[                    ]      31  INVOKESTATIC   java/lang/System.arraycopy
[                 L00]      32  ALOAD          0
[             L03 L00]      33  ALOAD          3
[                    ]      34  PUTFIELD       8
[                    ] L005:
                               ...............

[                    ]      45  RETURN
```

Figure 3.4: Example: intermediate instructions and stack contents

## 3.6 IA32 (x86, i386) code generator

### 3.6.1 Introduction

The IA32 architecture is the most important architecture on the desktop market. Since the current IA32 processors are getting faster and more powerful, the IA32 architecture also becomes more important in the low-end and mid-end server market. Major Java Virtual Machine vendors, like Sun or IBM, have highly optimized IA32 ports of their Virtual Machines, so it's fairly important for an Open Source Java Virtual Machine to have a good IA32 performance.

Porting CACAO to the IA32 platform was more effort than expected. CACAO was designed to run on RISC machines from ground up, so the whole code generation part has to be adapted. The first approach was to replace the simple RISC macros with IA32 code, but this turned out to be not successful. So new IA32 code generation macros were written, with no respect to the old RISC macros.

Some smaller problems occured since the IA32 port was the first 32 bit target platform, like segmentation faults due to heap corruption, which turned out to be a simple `for` loop bug only hit on 32 bit systems. Most of the CACAO system already was *32-bit-ready*, namely an architecture dependent `types.h` with definitions of the used datatypes and some feature flags, which features the processor itself natively supports. Most noticeable change was the `s8` and `u8` datatype, changed from `long` to `long long` to support 64 bit calculations.

### 3.6.2 Code generation

One big difference in writing the new code generation macros was, that the IA32 architecture is not a *load-store architecture* like the RISC machines, but the *machine instructions* can handle both *memory operands* and *register operands*. This led to a much more complicated handling of the various ICMDs. The typical handling of an ICMD on RISC machines looks like this (on the example of the integer add ICMD):

```
case ICMD_IADD:
    var_to_reg_int(s1, src->prev, REG_ITMP1);
    var_to_reg_int(s2, src, REG_ITMP2);
    d = reg_of_var(iptr->dst, REG_ITMP3);
```

```
            M_IADD(s1, s2, d);
            store_reg_to_var_int(iptr->dst, d);
            break;
```

This means loading the two *source operands* from memory to temporary registers, if necessary, getting a *destination register*, do the calculation and store the result to memory, if the destination variable resides in memory. If all operands are assigned to registers, only the calculation is done. This design also works on IA32 machines but leads to much bigger code size, reduces decoding bandwith and increases register pressure in the processor itself, which results in lower performance [Int03]. Thus CACAO uses all kinds of instruction types that are available and decide which one is used in some `if` statements:

```
if (IS_INMEMORY(iptr->dst)) {
    if (IS_INMEMORY(src) && IS_INMEMORY(src->prev)) {
        ...
    } else if (IS_INMEMORY(src) && !IS_INMEMORY(src->prev)) {
        ...
    } else if (!IS_INMEMORY(src) && IS_INMEMORY(src->prev)) {
        ...
    } else {
        ...
    }
} else {
    if (IS_INMEMORY(src) && IS_INMEMORY(src->prev)) {
        ...
    } else if (IS_INMEMORY(src) && !IS_INMEMORY(src->prev)) {
        ...
    } else if (!IS_INMEMORY(src) && IS_INMEMORY(src->prev)) {
        ...
    } else {
        ...
    }
}
```

For most ICMDs the generated code can be further optimized when one source variable and the destination variable share the same local variable.

To be backward compatible, mostly in respect of embedded systems, all generated code can be run on i386 compatible systems.

Another problem was the access to the functions' data segment. Since RISC platforms like Alpha and MIPS have a procedure vector register, for the IA32 platform there had to be implemented a special handling for accesses to the data segment, like `ICMD_PUTSTATIC` and `ICMD_GETSTATIC` instructions. The solution is like the handling of *jump references* or *check cast references*, which also have to be code patched, when the code and data segment are relocated. This means, there has to be an extra *immediate-to-register* move (`i386_mov_imm_reg()`) before every `ICMD_PUTSTATIC/ICMD_GETSTATIC` instruction, which moves the start address of the procedure, and thus the start address of the data segment, in one of the temporary registers (code snippet from `ICMD_PUTSTATIC`):

```
i386_mov_imm_reg(0, REG_ITMP2);
dseg_adddata(mcodeptr);
```

The `dseg_adddata()` call inserts the current postion of the code generation pointer into a datastructure which is later processed by `codegen_finish()`, where the final address of the data segment is patched.

### 3.6.3 Constant handling

Unlike RISC machines the IA32 architecture has *immediate move* instructions which can handle the maximum bitsize of the registers. Thus the IA32 port of CACAO does not have to load big constants indirect from the data segment, which means a *memory load* instruction, but can move 32 bit constants *inline* into their destination registers.

```
i386_mov_imm_reg(0xcafebabe, REG_ITMP1);
```

For constants bigger than 32 bits up to 64 bits, we split the move up into two immediate move instructions.

### 3.6.4 Calling conventions

The normal calling conventions of the IA32 processor is passing all function arguments on the stack [Int02]. The store size on the stack depends on the data type (see table 3.4).

This convention has been changed for CACAO in a way, that each datatype uses always 8 bytes on the stack. due to this fact after calling the function

| JAVA Data Type | Bytes |
|---|---|
| `boolean`<br>`byte`<br>`char`<br>`short`<br>`int`<br>`void`<br>`float` | 4 |
| `long`<br>`double` | 8 |

Table 3.4: IA32 calling convention stack store sizes

```
void sub(int i, long l, float f, double d);
```

the stack layout looks like in figure 3.5.

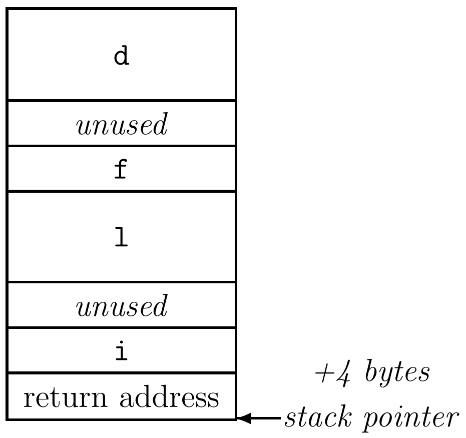

Figure 3.5: CACAO IA32 stack layout after function call

If the function passes a 32-bit variable, CACAO just push 4 bytes onto the stack and leave the remaining 4 bytes untouched. This does not make any problems since CACAO does not read a 64-bit value from a 32-bit location. Passing a 64-bit value is straightforward.

With this adaptation, it is possible to use the *stack space allocation algorithm* without any changes. The drawback of this decision is, that all arguments of a native function calls have to be copied into a new stack frame and the memory footprint is slightly bigger.

But calling a native function always means a stack manipulation, because the *JNI environment*, and additionally for `static` functions the *class pointer*, have to be stored in front of the function parameters. So this negligible.

For some `BUILTIN` functions there are assembler function counterparts, which copy the 8 byte parameters in their correct size in a new stack frame. But this only affects `BUILTIN` functions with more than one function parameter. To be precise, two functions, namely `arrayinstanceof` and `newarray`. So this is not a big speed impact.

Return parameters are stored in different places, this depends on the return type of the function:

- `boolean`, `byte`, `char`, `short`, `int`, `void`: return value resides in `%eax` (`REG_RESULT`)
- `long`: return value is split up onto the register pair `%edx:%eax` (`REG_RESULT2:REG_RESULT`, high 32-bit in `%edx`, low 32-bit in `%eax`)
- `float`, `double`: return value resides in the *top of stack* element of the *floating point unit* stack (`st(0)`, described in more detail in section 3.6.7)

### 3.6.5 Register allocation

Register usage was another problem in porting the CACAO to IA32. An IA32 processor has 8 integer general-purpose registers (GPR), of which one is the *stack pointer* (SP) and thus can not be used for arithmetic instructions. From the remaining 7 registers, in *worst-case instructions* like `CHECKCAST` or `INSTANCEOF`, 3 temporary registers need to be reserved for storing temporary values. Due to this fact there are 4 integer registers available for arithmetic operations.

CACAO uses `%ebp`, `%esi`, `%edi` as callee saved registers, which are callee saved registers in the IA32 ABI and `%ebx` as scratch register, which is also a callee saved register in the IA32 ABI. The remaining `%eax`, `%ecx` and `%edx` registers are used as the previously mentioned temporary registers.

The register allocator itself is very straightforward, this means, it does neither *linear scan* nor any other analyse of the methods variables, but allocates registers for the local variables in order as they are defined—*first-come-first-serve*. This may result in a fairly good register allocation on RISC machines, as there are almost always enough registers available for the functions local variables, but can result in a really bad allocation on IA32 processors.

So the first step to make the IA32 port more competitive with Sun's or IBM's JVM would be to rewrite the register allocator.

Only small register allocator changes were necessary for the IA32 port. Since CACAO—on the IA32 architecture—stores all `long` variables, because of lack of integer general-purpose registers, in memory locations (described in more detail in section 3.6.6) the register allocator has to be adapted to support this feature. This means all `long` variables are assigned to stack locations and tagged with the `INMEMORY` flag.

### 3.6.6 Long arithmetic

Unlike the PowerPC port, the IA32 port cannot easily store `long`'s in two 32-bit integer registers, since there are too little of them. Maybe this could bring a speedup, if the register allocator would be more intelligent or in leaf functions which have only `long` variables. But this is not implemented yet. So, the current approach is to store all `long`'s in memory, this means they are always spilled.

Nearly all `long` instructions are inline, except two of them: `ICMD_LDIV` and `ICMD_LREM`. These two are computed via `BUILTIN` calls. It would also be possible to inline them, but the code size would be too big and the latency of the `idiv` machine instruction is so high, that the function calls are negligible.

The IA32 processor has some machine instructions which are specifically designed for 64-bit operations. With them several 64-bit integer arithemtic operations can be implemented very efficiently [Adv02]. Some of them are

- `cltd` — Convert Signed Long to Signed Double Long
- `adc` — Integer Add With Carry
- `sbb` — Integer Subtraction With Borrow

Thus some of the 64-bit calculations like `ICMD_LADD` or `ICMD_LSUB` could be executed in two instructions, if both operand would reside in registers. But this does not apply to CACAO, yet.

The generated machine code of intermediate commands which operate on `long` variables instructions always operate on 64-bit, even if it is not necessary. The dependency information that would be required to just operate on the lower or upper half of the `long` variable, is not generated by CACAO.

| st(x) | FPU Data Register Stack |
|---|---|
| 0 | TOS (Top Of Stack) |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

Table 3.5: x87 FPU Data Register Stack

### 3.6.7 Floating point arithmetic

Since the i386, with it's i387 extension or the i486, the IA32 processor has a *floating point unit* (FPU). This FPU is implemented as a stack with 8 elements (see table 3.5).

This stack is designed to wrap around if values are loaded to the *top of stack* (TOS). For this reason, it has a special register which points to the TOS. This pointer is increased if a load instruction to the TOS is executed and decreased for a store from the TOS.

At first sight, the stack design of the FPU is perfect for the stack based design of a Java Virtual Machine. But there is one problem. The JVM stack has no fixed size, so it can grow up to much more than 8 elements and this simply results in an stack wrap around and thus an stack overflow. For this reason it it necessary to implement an *stack-element-to-register-mapping.*

A very basic design idea, is to define a pointer to the current TOS offset (`fpu_st_offset`). With this offset the current register position in the FPU stack of an arbitrary register can determined. From the 8 stack elements the last two ones (`st(6), st(7)`) are reserved, so two memory operands can be loaded onto the stack and to preform an arithmetic operation. Most IA32 floating point arithmetic operations have an *do arithmetic and pop one element* version of the instruction, that means the float arithmetic is done and the TOS element is poped off. The remaining stack element has the result of the calculation. On the example of the `ICMD_FADD` intermediate command with two memory operands, it looks like this:

```
/* load 1st operand in st(0), increase fpu_st_offset */
```

| Precision | PC Field |
|---|---|
| single-precision (32 bit) | 00B |
| reserved | 01B |
| double-precision (64 bit) | 10B |
| double extended-precision (80 bit) | 11B |

Table 3.6: Precision Control Field (PC)

```
var_to_reg_flt(s1, src->prev, REG_FTMP1);

/* load 2nd operand in st(0), increase fpu_st_offset */
var_to_reg_flt(s2, src, REG_FTMP2);

/* add 2 uppermost elements st(1) = st(1) + st(0), pop st(0) */
i386_faddp();

/* decrease fpu_st_offset from previous pop */
fpu_st_offset--;

/* store result -- decrease fpu_st_offset */
store_reg_to_var_flt(iptr->dst, d);
```

This mapping works very good with *scratch registers* only. Defining *callee saved float registers* makes some problemes since the IA32 ABI has no callee saved float registers. This would need a special handling in the *native stub* of a native function, namely saving the registers and cleaning the whole FPU stack, because a C function expects a clear FPU stack.

Basically the IA32 FPU can handle 3 float data types:

- single-precision (32 bit)
- double-precision (64 bit)
- double extended-precision (80 bit)

The FPU has a 16 bit *control register* which has a *precision control field* (PC) and a *rounding control field* (RC), each of 2 bit length (see table 3.6 and 3.7).

| Rounding Mode | RC Field |
|---|---|
| round to nearest (even) | 00B |
| round down (toward -$\infty$) | 01B |
| round up (toward +$\infty$) | 10B |
| round toward zero (truncate) | 11B |

Table 3.7: Rounding Control Field (RC)

The internal data type used by the FPU is always the *double extended-precision* (80 bit) format. Therefore implementing a IEEE 754 compliant floating point code on IA32 processors is not trivial.

Correct rounding to *single-precision* or *double-precision* is only done if the value is stored into memory. This means for certain instructions, like `ICMD_FMUL` or `ICMD_FDIV`, a special technique called *store-load*, has to be implemented [OKN03]. This technique is in fact very simple but takes two memory accesses more for this instruction.

For single-precision floats the *store-load* code could looks like this:

```
/* store single-precision float to stack */
i386_fstps_membase(REG_SP, 0);

/* load single-precision float from stack */
i386_flds_membase(REG_SP, 0);
```

Another technique which has to be implemented to be IEEE 754 compliant, is *precision mode switching*. Mode switching on the IA32 processor is made with the `fldcw`—load control word—instruction. A `fldcw` instruction has a quite large overhead, so frequently mode switches should be avoided. For this technique there are two different approaches:

- **Mode switch on float arithmetic** — switch the FPU on initialization in one precision mode, mostly *double-precision mode* because `double` arithmetic is more common. With this setting `double`s are calculated correctly. To handle `float`s in this approach, the FPU has to be switched into *single-precision mode* before each `float` instruction and switched back afterwards. Needless to say, that this is only useful, if `float` arithmetic is sparse. For a simple calculation like

```
float f = 1.234F;
float g = 2.345F;
float e = f * f + g * g;
```

the generated ICMD's look like this (with comments where precision mode switches take place):

```
...
<fpu in double-precision mode>
FLOAD        1
FLOAD        1
<switch fpu to single-precision mode>
FMUL
<switch fpu to double-precision mode>
FLOAD        2
FLOAD        2
<switch fpu to single-precision mode>
FMUL
<switch fpu to double-precision mode>
<switch fpu to single-precision mode>
FADD
<switch fpu to double-precision mode>
FSTORE       3
...
```

For this corner case situation the mode switches are extrem, but the example should demonstrate how this technique works.

- **Mode switch on float data type change** — switch the FPU on initialization in on precision mode, like before, in *double-precision mode*. But the difference on this approach is, that the precision mode is only switched if the float data type is changed. That means if your calculation switches from `double` arithmetic to `float` arithmetic or backwards. This technique makes much sense due to the fact that there are always a bunch of instructions of the same data type in one row in a normal program. Now the same example as before with this approach:

```
...
```

```
<fpu in double-precision mode>
FLOAD       1
FLOAD       1
<switch fpu to single-precision mode>
FMUL
FLOAD       2
FLOAD       2
FMUL
FADD
FSTORE      3
...
```

After this code sequence the FPU is in *single-precision mode* and if a function return would occur, the caller function would not know which FPU mode is currently set. One solution would be to reset the FPU to *double-precision mode* on a function return, if the actual mode is *single-precision*.

Each of these FPU techniques described have been implemented in CACAO's IA32 port, but the results were not completly IEEE 754 compliant. So the CACAO developer team decided to be on the safe side and to store all float variables in memory, until we have found a solution which is fast and 100% compliant.

### 3.6.8 Exception handling

The exception handling for the IA32 architecture is implemented as intended to be for CACAO. To handle the common and unexpected, but often checked, `java.lang.NullPointerException` very fast, CACAO uses *hardware null-pointer checking*. That means a signal handler for the `SIGSEGV` operating system signal is installed. If the signal is hit, the CACAO signal handler forwards the exception to CACAO's internal exception handling system. So if an instruction tries to access the memory at address `0x0`, a `SIGSEGV` signal is raised because the memory page is not read or writeable. After the signal is hit, the handler has to be reinstalled, so that further exceptions can be catched. This is done in the handler itself.

The `SIGSEGV` handler is used on any architecture CACAO has been ported to. Additionally on the IA32 architecture a handler for the `SIGFPE` signal is installed. With this handler a `java.lang.ArithmeticException`'s for integer */ by zero* can be catched in hardware and there is no need to write helper functions, like `asm_builtin_idiv`, which check the division operands as this is done for the Alpha or MIPS port.

## 3.7 AMD64 (x86_64) code generator

### 3.7.1 Introduction

The AMD64 [AMD04b] architecture, formerly known as x86_64, is an improvement of the Intel IA32 architecture by AMD—Advanced Micro Devices [AMD04a]. The extraordinary success of the IA32 architecture and the upcoming memory address space problem on IA32 high-end servers, led to a special design decision by AMD. Unlike Intel, with it's completely new designed 64-bit architecture—IA64—AMD decided to extend the IA32 instruction set with a new 64-bit instruction mode.

Due to the fact that the IA32 instructions have no fixed length, like this is the fact on RISC machines, it was easy for AMD to introduce a new *prefix byte* called `tablerexprefixbytefields`. The *REX prefix* enables the 64-bit operation mode of the following instruction in the new *64-bit mode* of the processor.

A processor which implements the AMD64 architecture has two main operating modes:

- Long Mode
- Legacy Mode

In the *Legacy Mode* the processor acts like an IA32 processor. Any 32-bit operating system or software can be run on these type of processors without changes, so companies running IA32 servers and software can change their hardware to AMD64 and their systems are still operational. This was the main intention for AMD to develop this architecture. Furthermore the *Long Mode* is split into two coexistent operating modes:

- 64-bit Mode
- Compatibility Mode

The *64-bit Mode* exposes the power of this architecture. Any memory operation now uses 64-bit addresses and ALU instructions can operate on 64-bit operands. Within *Compatibility Mode* any IA32 software can be run under the control of 64-bit operating system. This, as mentioned before, is yet another point for companies to change their hardware to AMD64. So

| Mnemonic | Bit Position | Definition |
|---|---|---|
| - | 7-4 | 0100 |
| REX.W | 3 | 0 = Default operand size<br>1 = 64-bit operand size |
| REX.R | 2 | 1-bit (high) extension of the ModRM *reg* field, thus permitting access to 16 registers. |
| REX.X | 1 | 1-bit (high) extension of the SIB *index* field, thus permitting access to 16 registers. |
| REX.B | 0 | 1-bit (high) extension of the ModRM *r/m* field, SIB *base* field, or opcode *reg* field, thus permitting access to 16 registers. |

Table 3.8: REX Prefix Byte Fields

their software can be slowly migrated to the new 64-bit systems, but not every type of software is faster in 64-bit code. Any memory address fetched or stored into memory needs to transfer now 64-bits instead of 32-bits. This means twice as much memory transfer as on IA32 machines.

Another crucial point to make the AMD64 architecture faster than IA32, is the limited number of registers. Any IA32 architecture, from the early *i386* to the newest generation of *Intel Pentium 4* or *AMD Athlon*, has only 8 general-purpose registers. With the *REX prefix*, AMD has the ability to increase the amount of accessible registers by 1 bit. This means in *64-bit Mode* 16 general-purpose registers are available. The value of a *REX prefix* is in the range `40h` through `4Fh`, depending on the particular bits used (see table 3.8).

### 3.7.2 Code generation

AMD64 code generation is mostly the same as on IA32. All new 64-bit instructions can handle both *memory operands* and *register operands*, so there is no need to change the implementation of the IA32 ICMDs.

Much better code generation can be achieved in the area of *long arithmetic*. Since all 16 general-purpose registers can hold 64-bit integer values, there is no need for special long handling, like on IA32 were we stored all long varibales in memory. As example a simple `ICMD_LADD` on IA32, best case shown for AMD64 — `s1 == d`:

```
i386_mov_membase_reg(REG_SP, s1 * 8, REG_ITMP1);
i386_alu_reg_membase(I386_ADD, REG_ITMP1, REG_SP, d * 8);
i386_mov_membase_reg(REG_SP, s1 * 8 + 4, REG_ITMP1);
i386_alu_reg_membase(I386_ADC, REG_ITMP1, REG_SP, d * 8 + 4);
```

First memory operand is added to second memory operand which is at the same stack location as the destination operand. This means, there are four instructions executed for one `long` addition. If we would use registers for `long` variables we could get a *best-case* of two instructions, namely *add* followed by an *adc*. On AMD64 we can generate one instruction for this addition:

```
x86_64_alu_reg_reg(X86_64_ADD, s1, d);
```

This means, the AMD64 port is *four-times* faster than the IA32 port (maybe even more, because we do not use memory accesses). Even if we would implement the usage of registers for `long` variables on IA32, the AMD64 port would be at least twice as fast.

To be able to use the new 64-bit instructions, we need to prefix nearly all instructions—some instructions can be used in their 64-bit mode without escaping—with the mentioned *REX prefix* byte. In CACAO we use a macro called

```
x86_64_emit_rex(size,reg,index,rm)
```

to emit this byte. The names of the arguments are respective to their usage in the *REX prefix* itself (see table 3.8).

The AMD64 architecture introduces also a new addressing method called *RIP-relative addressing*. In 64-bit mode, addressing relative to the contents of the 64-bit instruction pointer (program counter) — called *RIP-relative addressing* or *PC-relative addressing* — is implemented for certain instructions. In this instructions, the effective address is formed by adding the displacement to the 64-bit `RIP` of the next instruction. With this addressing mode, we can replace the IA32 method of addressing data in the method's data segment. Like in the `ICMD_PUTSTATIC` instruction, the IA32 code

```
a = dseg_addaddress(&(((fieldinfo *) iptr->val.a)->value));
i386_mov_imm_reg(0, REG_ITMP2);
dseg_adddata(mcodeptr);
i386_mov_membase_reg(REG_ITMP2, a, REG_ITMP2);
```

can be replaced with the new *RIP-relative addressing* code

```
a = dseg_addaddress(&(((fieldinfo *) iptr->val.a)->value));
x86_64_mov_membase_reg(RIP, -(((s8) mcodeptr + 7) - (s8) mcodebase) + a,
```

So we can save one instruction on the read or write of an static variable. The additional offset of `+ 7` is the code size of the instruction itself. The fictive register `RIP` is defined with

```
#define RIP    -1
```

Thus we can determine the special *RIP-relative addressing* mode in the code generating macro `x86_64_emit_membase(basereg,disp,dreg)` with

```
if ((basereg) == RIP) {
    x86_64_address_byte(0,(dreg),RBP);
    x86_64_emit_imm32((disp));
    break;
}
```

and generate the *RIP-relative addressing* code. As shown in the code sample, it's an special encoding of the *address byte* with `mod` field set to zero and `RBP` (`%rbp`) as baseregister.

## 3.7.3 Constant handling

As on IA32, the AMD64 code generator can use *immediate move* instructions to load integer constants into their destination registers. The 64-bit extensions of the AMD64 architecture can also load 64-bit immediates inline. So loading a `long` constant just uses one instruction, despite of two instructions on the IA32 architecture. Of course the AMD64 code generator uses the *move long* (`movl`) instruction to load 32-bit `int` constants to minimize code size. The `movl` instruction clears the upper 32-bit of the destination register.

```
case ICMD_ICONST:
        ...
        x86_64_movl_imm_reg(cd, iptr->val.i, d);
        ...
```

| JAVA Data Type | Bytes |
|---|---|
| `boolean`<br>`byte`<br>`char` | 1 |
| `short` | 2 |
| `int`<br>`float` | 4 |
| `long`<br>`double`<br>`void` | 8 |

Table 3.9: JAVA Data Type sizes on AMD64

```
case ICMD_LCONST:
        ...
        x86_64_mov_imm_reg(cd, iptr->val.l, d);
        ...
```

`float` and `double` values are loaded from the data segment via the *move doubleword or quadword* (`movd`) instruction with *RIP-relative addressing.*

### 3.7.4 Calling conventions

The AMD64 calling conventions are described here [HJM04]. CACAO uses a subset of this calling convention, to cover its requirements. CACAO just needs to pass the JAVA data types to called functions, no other special features are required. The byte sizes of the JAVA data types on the AMD64 port are shown in table 3.9.

#### Integer arguments

The AMD64 architecture has 6 integer argument registers. The order of the argument registers is shown in table 3.10.

As on RISC machines, the remaining integer arguments are passed on the stack. Each integer argument, regardless of which integer JAVA data type, uses 8 bytes on the stack.

| Register | Argument Register |
|---|---|
| `%rdi` | 1st |
| `%rsi` | 2nd |
| `%rdx` | 3rd |
| `%rcx` | 4th |
| `%r8` | 5th |
| `%r9` | 6th |

Table 3.10: AMD64 Integer Argument Register

Integer return values of any integer JAVA data type are stored in `REG_RESULT`, which is `%rax`.

## Floating-point arguments

The AMD64 architecture has 8 floating point argument registers, namely `%xmm0` through `%xmm7`. `%xmm` registers are 128-bit wide floating point registers on which SSE and SSE2 instructions can operate. Remaining floating point arguments are passed, like integer arguments, on the stack using 8 bytes per argument, regardless to the floating-point JAVA data type.

Floating point return values of any floating-point JAVA data type are stored in `%xmm0`.

As shown, the calling conventions for the AMD64 architecture are similar to the calling conventions of RISC machines, which allows to use CACAOs *register allocator algorithm* and *stack space allocation algorithm* without any changes.

Calling native functions means register moves and stack copying like on RISC machines. This depends on the count of the arguments used for the called native function. For non-static native functions the first integer argument has to be the JNI environment variable, so any arguments passed need to be shifted by one register, which can include creating a new stackframe and storing some arguments on the stack. Additionally for static native functions the class pointer of the current objects' class is passed in the 2nd integer argument register. This means that the integer argument registers need to be shifted by two registers.

One difference of the AMD64 calling conventions to RISC type machines, like Alpha or MIPS, is the allocation of integer and floating point argument registers with mixed integer and floating point arguments. Assume a function like this:

```
void sub(int a, float b, long c, double d);
```

On a RISC machine, like Alpha, we would have an argument register allocation like in figure 3.6. `a?` represent integer argument registers and `fa?` floating point argument registers.

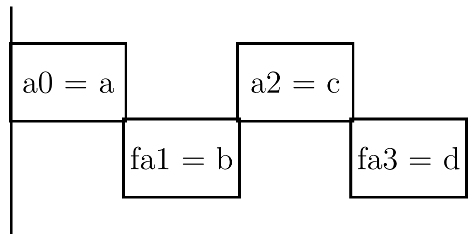

Figure 3.6: Alpha argument register usage for `void sub(int a, float b, long c, double d);`

On AMD64 the same function call would look like in figure 3.7.

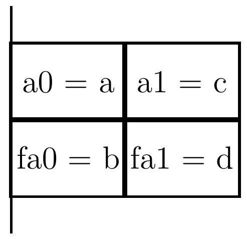

Figure 3.7: AMD64 argument register usage for `void sub(int a, float b, long c, double d);`

The register assigment would be `a0 = %rdi`, `a1 = %rsi`, `fa0 = %xmm0` and `fa1 = %xmm1`. This special usage of the argument registers required some changes in the argument register allocation algorithm for function calls during stack analysis and some changes in the code generator itself.

| Register | Usage | Callee-saved |
|---|---|---|
| `%rax` | return register, reserved for code generator | no |
| `%rcx` | 4th argument register | no |
| `%rdx` | 3rd argument register | no |
| `%rbx` | temporary register | no |
| `%rsp` | stack pointer | yes |
| `%rbp` | callee-saved register | yes |
| `%rsi` | 2nd argument register | no |
| `%rdi` | 1st argument register | no |
| `%r8` | 5th argument register | no |
| `%r9` | 6th argument register | no |
| `%r10 - %r11` | reserved for code generator | no |
| `%r12 - %r15` | callee-saved register | yes |
| `%xmm0` | 1st argument register, return register | no |
| `%xmm1 - %xmm7` | argument registers | no |
| `%xmm8 - %xmm10` | reserved for code generator | no |
| `%xmm11 - %xmm15` | temporary registers | no |

Table 3.11: AMD64 Register usage in CACAO

### 3.7.5 Register allocation

As mentioned in the introduction, the AMD64 architecture has 16 integer general-purpose registers and 16 floating-point registers. One integer general-purpose register is reserved for the *stack pointer*—namely `%rsp`—and thus cannot be used for arithmetic instructions. The register usage as used in CACAO is shown in table 3.11.

There is only one change to the original AMD64 *application binary interface* (ABI). CACAO uses `%rbx` as temporary register, while the AMD64 ABI uses the `%rbx` register as callee-saved register. So CACAO needs to save the `%rbx` register when a JAVA method is called from a native function, like a JNI function. This is done in `asm_calljavafunction` located in `jit/x86_64/asmpart.S`.

In adapting the register allocator there was a problem concerning the order of the integer argument registers. The order of the first four argument register is inverted. This fact can be seen in table 3.11 which is ordered ascending by the processors' internal register numbers. That means the ascending search algorithm for argument registers in the register allocator would allocate the first four argument registers in the wrong direction. So there is a little hack implemented in CACAOs register allocator to handle this fact. After searching the register definition array for the argument registers, the first four argument registers are interchanged in their array. This is done by a simple code sequence (taken from `jit/reg.inc`):

```
/*
 * on x86_64 the argument registers are not in
 * ascending order
 * a00 (%rdi) <-> a03 (%rcx) and
 * a01 (%rsi) <-> a02 (%rdx)
 */
n = r->argintregs[3];
r->argintregs[3] = r->argintregs[0];
r->argintregs[0] = n;

n = r->argintregs[2];
r->argintregs[2] = r->argintregs[1];
r->argintregs[1] = n;
```

### 3.7.6 Floating-point arithmetic

The AMD64 architecture has implemented two sets of floating-point instructions:

- x87 (i387)
- SSE/SSE2

The x87 *floating-point unit* (FPU) implementation is completely compatible to the IA32 implementation, since the i386 with its i387 coproccessor, with all the advantages and drawbacks, like the 8 slot FPU stack.

The SSE/SSE2 technique is taken from the newest generation of Intel processors, introduced with Intel's Pentium 4, and can process scalar 32-bit `float` values and scalar 64-bit `double` values in the 128-bit wide `xmm` floating-point registers. While SSE instructions operate on 32-bit `float` values, SSE2 is responsible for 64-bit `double` values. In CACAO we implemented the JAVA floating-point instructions using SSE/SSE2, because SSE/SSE2 is much easier to use and should be the technique of the future. In some areas SSE/SSE2 is slower than the old x87 implementation, even on the new designed AMD64 architecture, but SSE/SSE2 offers 16 floating-point registers, which should speed up daily JAVA floating-point calculations. Another big advantage of SSE/SSE2 to x87 is the missing *single-double precision-rounding* problem, as described in detail in the "IA32 code generator" section 3.6. With SSE/SSE2 the 32-bit `float` and 64-bit `double` arithmetic is calculated and rounded completely IEEE 754 compliant, so no further adjustments need to take place to fullfil JAVAs floating-point requirements.

In floating-point value to integer value conversions a JVM has to check for corner cases as described in the JVM specification. This is done via a simple inline integer compare of the integer result value and a call to special assembler wrapper functions for builtin calls, like `asm_builtin_f2i` for `ICMD_F2I` — `float` to `int` conversion. These corner cases are then computed in a builtin C function with respect to all special cases like *Infinite* or *NaN* values.

### 3.7.7 Exception handling

Since the AMD64 architecture is just an extension to the IA32 architecture, an AMD64 processor itself raises the same signals as an IA32 processor, so we can catch the same signals in our own signal handlers. This includes the signals `SIGSEGV` and `SIGFPE`.

When a signal of this type is raised and the signal hits our signal handler, we reinstall the handler, create a new exception object and jump to a—in assembler—written exception handling code. The difference to the exception handling code of RISC machines, is the fact that RISC machines have a *procedure vector* (PV) register. So it's easy to find the methods' data segment, which starts at the PV growing down to smaller addresses like a stack. For the IA32 and AMD64 architecture we had to implement a *method tree* which contains the start *program counter* (PC) and the end PC for every single JAVA method compiled in CACAO, to find for any exception PC the corresponding method and thus the PV. We need the data segment for the methods' exception table (for a detailed description see section "Exception handling").

We use `SIGSEGV` for *hardware null-pointer checking*, so we can handle this common exception as fast as possible in CACAO. The signal handler creates a `java.lang.NullPointerException`.

`SIGFPE` is used to catch integer division by zero exceptions in hardware. The signal handler generates a `java.lang.ArithmeticException` with `/ by zero` as detail message.

Both exceptions are handled in hardware by default, but they can also be catched in software when using CACAOs commandline switch `-softnull`. On the RISC ports only the *null-pointer exception* is checked in software when using this switch, but on IA32 and AMD64 both are checked, `SIGSEGV` and `SIGFPE`.

# Chapter 4

# Evaluation

## 4.1 Environment

For the measurements the current development version of CACAO at the time of writing this document has been used. The currently integrated GNU Classpath is version 0.10. The measurements were performed on an Alpha 21164 600 Mhz with 512 MB physical memory running GNU/Linux. This rather slow machine was choosen to minimize the measuring error in the rather short loading times and overall run times of the benchmarked programs.

The second test machine was an Intel Penitum 4 2.26 GHz with 1 GB physical memory. On this machine comparisons between different Java Virtual Machine were performed because most Just-In-Time compilers are available for the IA32 architecture.

The third test machine was a dual AMD Opteron 246 2 GHz with 2 GB physical memory. This machine was used to make some further constant array store benchmarks.

All benchmark class files and the GNU classpath files were stored plainly in the filesystem and were not packed in a zip or jar file.

The following sections describe in more detail the different optimizations shown throughout this document and implemented in CACAO. Section 4.2 describes the speedup of CACAO startup times between *eager class loading* and *lazy class loading*. Section 4.3 shows the speedup gained by the implementation of *constant array stores*. Section 4.4 shows some run time results of CACAO in comparison with other well-known Java Virtual Machines.

| Benchmark | | eager loading | lazy loading | shortage/ speedup |
|---|---|---|---|---|
| `HelloWorld` | class loads | 513 | 121 | 4.23 |
| | loaded methods | 5236 | 1301 | 4.02 |
| | class loading time (in sec) | 0.510 | 0.130 | 3.92 |
| | overall run time (in sec) | 0.667 | 0.248 | 2.69 |
| `kjc` | class loads | 985 | 496 | 1.99 |
| | loaded methods | 9222 | 4917 | 1.88 |
| | class loading time (in sec) | 1.125 | 0.205 | 5.48 |
| | overall run time (in sec) | 2.185 | 1.675 | 1.30 |
| `javac` | class loads | 710 | 314 | 2.26 |
| | loaded methods | 7438 | 3770 | 1.97 |
| | class loading time (in sec) | 0.912 | 0.539 | 1.69 |
| | overall run time (in sec) | 1.738 | 1.329 | 1.31 |

Table 4.1: Eager class loading vs. lazy class loading on Alpha

## 4.2 Eager class loading vs. lazy class loading

The startup times of the CACAO Java Virtual Machine with lazy class loading should be improved significantly. Since the current implementation of the eager class loading algorithm still has some minor problems concerning the bootstrapping of `java.lang.Object`, CACAO was compiled with the configure option `--disable-threads`. Without threads the eager class loading algorithm works without a glitch on every target platform.

Table 4.1 shows the difference of some simple benchmarks between eager class loading and lazy class loading.

As the results show the number of loaded classes is reduced seriously. In a simple `HelloWorld` the loaded classes are reduced by a mean factor of 4.23. This results in a speedup of the class loading time by a factor of 3.92. The overall run time speedup of the `HelloWorld` program is not as high as the class loading speedup, because the JIT compilation time and the code execution times are of course the same as with eager class loading but nonetheless it is a remarkable speedup by a factor of 2.69.

The `kjc` benchmark is a simple compilation of a `HelloWorld.java` source file with the Kopi Java Compiler [Lac01] [KJC04] version 2.1B. The benchmark was invoke via

```
cacao -time -stat at.dms.kjc.Main -O0 HelloWorld.java
```

As the table shows the class load shortage from `kjc` is by a factor 2 smaller than from `HelloWorld`. This is due to the fact that the Kopi compiler classes itself are loaded anyway and these classes mostly reference to it's own compiler classes. This fact also reflects in the number of loaded methods. On the other hand the class loading time is decreased dramatically. The class loading is nearly 5.5 times faster than with eager class loading. The overall run time is *only* decreased by a factor of 1.3 because of the rather long run time of the Kopi compiler itself.

The `javac` benchmark was performed with the classfiles from Sun's J2SE 1.4.2. The benchmark was invoked via

```
cacao -time -stat com.sun.tools.javac.Main \
    -g:none HelloWorld.java
```

The results are quite comparable to the `kjc` results. The Java compiler benchmarks were choosen because they are very complex, use a fairly huge amount of class files and should have a small startup and execution time.

It's not quite obvious why the lazy class loading time in the `kjc` benchmark is that much faster than with eager class loading. This fact has to be researched in more detail.

This overall speedup is applicable to all architectures CACAO has been ported to. But it is best measured on this rather slow Alpha machine which has been used.

## 4.3 Constant array stores

Some architectures support storing of immediate values directly into memory without loading them first into registers, mainly IA32 and AMD64. This *constant array store* optimization has been implemented into CACAO. This short code sample

```
no_prime[i] = false;
```

taken from the `sieve` benchmark is translated into four CACAO intermediate commands:

| Benchmark | Architecture | w/o constant array stores overall run time (in sec) | w/ constant array stores overall run time (in sec) | speedup |
|---|---|---|---|---|
| `sieve` | Alpha | 8.877 | 8.462 | 1.05 |
| | IA32 | 2.500 | 1.870 | 1.34 |
| | AMD64 | 1.636 | 1.575 | 1.04 |

Table 4.2: Constant array stores

```
[          101]        0  ALOAD         1
[      103 101]        1  ILOAD         3
[  t10 103 101]        2  ICONST        0
[             ]        3  IASTORE
```

The fact that the stored `int` value is constant can be used to combine the `ICMD_ICONST` and `ICMD_IASTORE` commands into one intermediate command—`ICMD_IASTORECONST`—which stores the constant value directly into the appropriate array position:

```
[          101]        0  ALOAD         1
[      103 101]        1  ILOAD         3
[             ]        2  IASTORECONST  0
```

Some RISC architectures have a special register which always contains the value zero—in CACAO called `REG_ZERO`. In the case shown above the optimization can also take place on this RISC machines like an Alpha or MIPS. `REG_ZERO` can be used as source register so no previous immediate load or register move needs to take place.

Table 4.2 shows the measured times on different architectures with and without constant array stores. The `sieve` benchmark was run without loop optimizations or removed bound checks. The invoked command was

```
cacao -time sieve 10000 10000
```

On the Alpha architecture the `sieve` algorithm was inverted, that means the prime array was initialized with `true` values and `false` was stored when a prime number was found. With this inversion CACAO was able to optimize the array store with the zero register.

| Benchmarks | Architecture | `ICMDs` | `ICMD_xASTORECONSTs` | Ratio |
|---|---|---|---|---|
| `kjc` | Alpha | 51,931 | 118 | 0.0023 |
| | IA32 | 51,846 | 1495 | 0.0288 |
| `javac` | Alpha | 39,980 | 38 | 0.0010 |
| | IA32 | 39,484 | 762 | 0.0193 |

Table 4.3: Number of constant array stores generated

Constant array stores speed up the execution of such a constant array store intensive program about 5%. The major speedup on the IA32 architecture is caused by the fact that the register allocator produces a worse allocation. The mostly used variable `i` is stored on the stack. With constant array stores the register pressure is reduced and one register is freed. This newly freed register is assigned to `i` and thus all variables now reside in a register. With a linear scan register allocator the measured speedup should be in the range of the AMD64.

Execution times of `kjc` and `javac` with and without constant array stores where to close and could be treated as error in measurement. Thus they are not shown in the table. Therefore table 4.3 shows the number of generated constant array store commands in the Java compilers. Again the compiled source file was `HelloWorld.java`.

On an IA32 architecture which supports *immediate-to-memory* stores the amount of generated commands is about 2%. On an Alpha the amount is so small that is it negligible.

## 4.4 CACAO vs. well-known Java Virtual Machines

The next measurement is a comparison between CACAO and well-known Java Virtual Machines with Just-In-Time compilers on the IA32 architecture. Used Java Virtual Machines are from major vendors as well as open source software. In detail the used Java Virtual Machines are:

- Sun J2SE 1.4.2_05, build 1.4.2_05-b04, mixed mode
- IBM J2SE 1.4.2, IBM build cxia321420-20040626
- Kaffe 1.1.4, Engine: Just-in-time v3

| Benchmark | Java Virtual Machine | overall run time (in sec) | speedup |
|---|---|---|---|
| `kjc` | CACAO | 0.35 | - |
| | Sun JVM (client) | 0.73 | 2.09 |
| | Sun JVM (server) | 1.38 | 3.94 |
| | IBM JVM | 0.96 | 2.74 |
| | Kaffe | 0.45 | 1.29 |
| `javac` | CACAO | 0.31 | - |
| | Sun JVM (client) | 0.81 | 2.61 |
| | Sun JVM (server) | 1.81 | 5.84 |
| | IBM JVM | 1.05 | 3.39 |
| | Kaffe | 0.41 | 1.32 |

Table 4.4: Java Virtual Machine comparison on IA32

Table 4.4 shows the overall run time results in seconds and the speedup of CACAO in comparison to the used Java Virtual Machine.

For this benchmarks CACAO has been recompiled with native threads enabled to get no performance gain. The benchmarks used were the same as used in the first measurement, that means a compilation of a `HelloWorld.java` source file with the specified compiler. The overall run times were measured with the `time(1)` system utility. All Java Virtual Machines and benchmarks have been invoked without any options, that means the default behaviour was measured. This decision was based on the fact that the invocation without options is the most likely case in everyday usage. The Sun JVM, as labeled in the table, was measured in both modes: client and server.

As the measurements show the program speedup when executed with CACAO in comparison to major vendor Java Virtual Machines are dramatically. Even compared to Kaffe the speedup is about 30%.

One interesting point not shown in the table is that every Java Virtual Machine has used nearly 100% of CPU time except Sun's. For both benchmarks the highest CPU usage in client mode was 85% and in server mode 70% although the machine had no load at all. The mean CPU load for client mode was 75% and for server mode about 45%. This means the Sun Java Virtual Machine would be in fact faster, but the results shown in the table are the times measured in practice.

# Chapter 5

# Related work

In the following related work to the different chapters of this document is presented. For better clarity the related work chapter is seperated into sections.

## 5.1 Class loader

The class loading algorithm of a Java Virtual Machine is completely described in [LY99]. Jensen et al. [JMT98] proposed a formalization of dynamic class loading in a Java Virtual Machine. This formal approach confirmed the type safety problem with class loaders. [BL99] presents the notion of class loaders and demonstrate some of their interesting uses.

Dynamic class loading is also implemented in other programming languages. In [Nor00] dynamic class loading in C++ is presented.

## 5.2 The Just-In-Time Compiler

The Java Programming Language has got more interesting with the introduction of Just-In-Time compilers. [LW00] shows some performance evaluation between Java and C++. [GSW$^+$] describes the design of the Jikes RVM—formerly known as Jalapẽno—optimizing compiler. This Java Virtual Machine uses the same compile approach as CACAO namely *compile-only* which means it just uses a Just-In-Time compiler and no interpreter.

## 5.3 IA32 code generator

Nearly all well-known Java Virtual Machines available have an optimizing Just-In-Time compiler for the IA32 architecture like Sun's JVM [Mic04], IBM's JVM [IBM04] or Kaffe [Wil97].

Porting a Java Virtual Machine to the IA32 architecture is always challenging. In [ABC$^+$02] the experiences of porting the Jikes RVM [AAB$^+$00] to Linux/IA32 are described.

## 5.4 AMD64 code generator

The AMD64 architecture is a reasonably young architecture, released in April 2003. At the writing of this document the only available 64-bit operating systems for AMD64 are GNU/Linux from different distributors, FreeBSD, NetBSD and OpenBSD. Microsoft Windows is not available yet, although it was announced to be released in the first half of 2004.

The first available 64-bit JVM for the AMD64 architecture was GCC's GCJ—The GNU Compiler for the Java Programming Language [GCJ04]. `gcj` itself is a portable, optimizing, ahead-of-time compiler for the JAVA Programming Language, which can compile:

- JAVA source code directly to native machine code
- JAVA source code to JAVA bytecode (class files)
- JAVA bytecode to native machine code

One part of the GCJ is `gij`, which is the JVM interpreter. Much of the porting effort for the *GNU Compiler Collection* to the AMD64 architecture was done by people working at SUSE [SUS04].

Long time no AMD64 JIT was available, till Sun [SUN04] released their AMD64 version of J2SE 1.4.2-rc1 for GNU/Linux by Blackdown [Bla03] in December 2003. At this time our AMD64 JIT was already working for months, but we were not able to release CACAO, because of the common status of CACAO to be a compliant JVM. The Sun JVM uses the HotSpot Server VM by default, the HotSpot Client VM is not available for AMD64 at this time.

The Kaffe [Wil97] JVM has ported their interpreter to the AMD64 architecture for GNU/Linux, but they still have no plans to port their JIT.

# Bibliography

[AAB$^+$00] B. Alpern, C. R. Attanasio, J. J. Barton, Burke Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, Ngo Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, ???? 2000.

[ABC$^+$02] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ngo Ngo. Experiences porting the Jikes RVM to Linux/IA32. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[Adv02] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. Publication No. 22007.

[AMD04a] Advanced Micro Devices. `http://www.amd.com/`, 2004.

[AMD04b] AMD64. `http://www.amd64.org/`, 2004.

[ATCL$^+$98] Ali-Reza Adl-Tabatabai, Michal Ciernak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Conference on Programming Language Design and Implementation*, volume 33(5) of *SIGPLAN*, pages 280–290, Montreal, 1998. ACM.

[BL99] Gilad Bracha and Sheng Liang. Dynamic class loading in the java, September 26 1999.

[Bla03] Blackdown.org. `http://www.blackdown.org/`, 2003.

[Cor98] The Standard Performance Evaluation Corporation. SPECjvm98: JVM Client98. `http://www.spec.org/osg/jvm98/`, 1998.

[Fou04] Free Software Foundation. GNU Classpath. `http://www.gnu.org/software/classpath/`, 2004.

[GCJ04] The GNU compiler for the Java programming language. `http://gcc.gnu.org/java/`, 2004.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[Gra97] Reinhard Grafl. CACAO: Ein 64Bit JavaVM Just-in-Time Compiler. Master's thesis, Technische Universität Wien, January 1997.

[GSW$^+$] David Grove, Harini Srinivasan, John Whaley, Jong deok Choi, Mauricio J. Serrano, Burke Burke, Michael Hind, Stephen Fink, Sreedhar Sreedhar, and Vivek Sarkar. The jalape no dynamic optimizing compiler for java.

[HJM04] Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement. `http://www.amd64.org/`, July 2004.

[IBM04] IBM. IBM Java Virtual Machine. `http://www.ibm.com/developerworks/java/jdk/`, 2004.

[Int02] Intel Cooperation, P.O. Box 7641 Mt. Prospect IL 60056-7641. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2002. Order Number: 245470-009.

[Int03] Intel Cooperation, `http://developer.intel.com`. *IA-32 Intel Architecture Optimization Reference Manual*, 2003. Order Number: 248966-009.

[JMT98] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 4–15. IEEE Computer Society Press, 1998.

[KJC04] KJC Kopi Java Compiler. `http://www.dms.at/kopi/`, 2004.

[Lac01] Martin Lackner. Extending Java. Master's thesis, Technische Universität Wien, May 2001.

[LW00] David D. Langan and Sean Wentworth. Performance evaluation: Java vs C++, December 19 2000.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[Mic04] Sun Microsystems. Sun Microsystems Java Virtual Machine. `http://java.sun.com/j2se/`, 2004.

[Nor00] James Norton. Dynamic class loading in C++. *Linux Journal*, 73:??–??, May 2000.

[OKN03] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. Optimizing Precision Overhead for x86 Processors. In *2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, volume August 1–2, pages 41–50, 2003.

[SUN04] Sun Microsystems. `http://www.sun.com/`, 2004.

[SUS04] SUSE Linux. `http://www.suse.com/`, 2004.

[Wil97] Tim Wilkinson. KAFFE: A free virtual machine to run Java code. `http://www.kaffe.org`, 1997.