Master's Thesis

SlimVM: A Small Footprint Java Virtual Machine for Connected Embedded Systems

Christoph Kerschbaumer

Institute for Technical Informatics, University of Technology, Graz
Head: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß
Assessor: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger

Department of Computer Science, University of California, Irvine
Assessor: Univ.-Prof. Dipl. Informatik-Ing. Dr.sc.tech. Michael Franz
Supervisor: Dipl.-Ing. Gregor Wagner

Graz, January 2009

Kurzfassung

Die Benutzung von Mobiltelefonen, PDAs und anderen mobilen Geräten ist in den letzten zehn Jahren drastisch angestiegen. Java, als eine der beliebtesten Ausführungsumgebungen, findet besonders häufig Verwendung auf solchen Systemen. Da mobile Geräte in ihrer Speicherkapazität stark limitiert sind, Java jedoch zum Ausführen eines Programms eine hohe Anzahl an Bibliotheken benötigt, ist es Ziel dieser Arbeit, den gesamten überflüssigen Code sowie Metainformationen auf dem mobilen Gerät zu entfernen und auf dem Server zu belassen. Genau auf diesem Ansatz beruht die Idee der Slim Virtual Machine.

Diese Arbeit präsentiert eine neue Generation von "permanent verbundenen mobilen Geräten", wobei der gesamte Code auf einem Netzwerk-Host verbleibt und von der Java Virtual Machine am Client partiell angefordert wird. Hierzu wird der gesamte Programm- und Bibliothekscode am Server analysiert, und nur der Code, der unmittelbar für die Ausführung des Programms benötigt wird, während der Laufzeit zum Client transferiert. Hierfür wird Java Bytecode am Server manipuliert und in Form von "fertig gelinkten" Blöcken an den Client geschickt, da diese die kleinste logische Einheit von Java Bytecode darstellen. Die daraus ergebenen Messungen zeigen, dass diese neu entwickelte Methode eine Reduzierung des Memory Footprints von bis zu 70% zulässt.

Schlüsselwörter: Java virtuelle Maschine, Optimierung, Codeoptimierung

Abstract

The usage of cellular phones, PDAs and other mobile devices has increased dramatically over the past ten years. Java is targeted to be one of the most popular execution environments on such systems. However, since mobile devices are inherently limited in terms of local storage capacity and Java requires large amounts of library code to be present on each client device, it is crucial to reduce the code footprint to ensure Java's success on such systems. The SlimVM approach is aiming at replacing all unnecessary code on mobile devices.

This thesis presents a solution for the next generation of mobile computing environments for persistent connected embedded systems where all code resides on a network host and is requested by the Java virtual machine on the client at run time. All application and library code is analyzed on the server prior to execution on the mobile device, and only code essential for execution is sent to the client on demand. Therefore, Java bytecode is manipulated and transferred to the client in the form of pre-linked basic blocks as this is the smallest granularity of Java bytecode. Measurements show a reduction of the memory footprint of up to 70%.

Keywords: Java virtual machine, optimization, connected embedded systems, code-size reduction

Acknowledgements

I am especially grateful to Christian Steger, Professor at the Institute for Technical Informatics, University of Graz, Austria and to Michael Franz, Professor at the Department of Computer Science, University of California, Irvine, USA for having facilitated this joint research project.

I express my gratitude to my supervisor Gregor Wagner. His guidance and support were invaluable for the completion of this thesis. Further, I would like to thank Christian Wimmer and Andreas Gal for their help and support during the development phase of the prototype implementation.

I also thank the entire SSLLAB group for having made my stay in California an unforgettable experience.

Finally, I thank my girlfriend Sabine, my sister Katja, and my parents Renate and Alfred for their support and encouragement during my studies.

Graz, January 2009 Christoph Kerschbaumer

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The first section of this chapter explains the motivation for this research project. Then, the history of the research collaboration between the Department of Computer Science, University of California Irvine and the Institute for Technical Informatics, University of Technology Graz is presented. Finally, this chapter gives an overview of the structure of this master's thesis.

1.1 Motivation

The usage of Java virtual machines has become very popular over the last decade. The idea that a program does not have to be compiled into object code for every single platform but can be executed by a virtual machine on the target device has started the triumphal procession of Java virtual machines. The use of such virtual machines is not restricted to personal computers and laptops, they are also very popular on mobile devices like cellular phones and PDAs. A lot of research has been done in this field, but all other implementations are limited by the completeness of the application and library code.

The goal of this research project is therefore to implement a client-server architecture for mobile devices where all application and library code resides on a network host and is transferred to the client only on demand. As mobile devices are inherently limited in terms of local storage capacity, a Java virtual machine with only a small memory footprint should be developed for the mobile device.

As bandwidth is a bottleneck for persistent connected embedded systems, a new way of transferring information to the mobile device should be developed, where only runtime critical information is extracted. All application and library code should be analyzed on the server and only code, necessarily needed for execution by the virtual machine should be requested from the server during runtime. Therefore, Java bytecode should be manipulated and transferred to the client in form of pre-linked basic blocks, the smallest possible logical collection of byte code instructions, as it is the smallest granularity of Java bytecode.

1.2 Project History

The research collaboration between the Institute for Technical Informatics, University of Technology Graz, Austria and the Department of Computer Science, University of California Irvine, USA started in 2006.

Christian Steger of the University of Technology Graz and Andreas Gal of the University of California Irvine met at a conference in Germany in 2005. During their conversation, Andreas Gal invited students from Graz over to California to join the research group of the *Secure Systems Laboratory* of Michael Franz in Irvine. Gregor Wagner and two other students from Graz took that opportunity and followed the invitation of Andreas Gal in early 2006 and spent one semester in Irvine to do research for their master's theses.

During this time, Gregor Wagner developed the first version of the SlimVM approach [Wag07] which he presented in his paper *SlimVM: Optimistic Partial Program Loading for Connected Embedded Java Virtual Machines* [WGF08] at the *Principles and Practice of Programming in Java* conference in Modena, Italy in 2008.

Currently Gregor Wagner is involved in a PhD program at the University of California Irvine and part of the research group of Michael Franz. When I started this project in March 2008, he became the supervisor of this master's thesis. With his guidance, a lot of new ideas to improve, and based on the original approach, we started to develope a new SlimVM from scratch. This time, JamVM is used for the prototype implementation, with the difference that JamVM is a Java virtual machine which supports the full Java specification. In contrast to the original SlimVM approach, we use the whole Java library instead of the small subset of the KVM [Sun99b] library.

1.3 Structure of this Master's Thesis

The remainder of this master's thesis is structured as follows and consists of seven chapters in which some chapters may depend on ideas or considerations of previous chapters. The main focal point is on the Chapters 4 and 5 which explain the design and the prototype implementation of this thesis.

Chapter 2 is meant to give an introduction and overview of the functionality of JamVM, a Java virtual machine with a small memory footprint. This chapter also gives an insight into the Java class file format and describes the bytecode format examplified by a simple factorial program code snippet. Furthermore, this chapter describes the *Byte Code Engineering Library* (BCEL) and the *Java Native Interface*.

Chapter 3 references the latest research proposals in terms of code size reduction, dead code elimination, and bytecode compression for Java class files and gives an overview of the state of the art.

Chapter 4 presents the principle approach and the design of the SlimVM approach. This chapter includes details of the client-server architecture and the communication between them. Furthermore, it includes some use cases of SlimVM and describes the Data Flow in the Slim virtual machine.

Chapter 5 gives an insight into the prototype implementation of SlimVM. Starting with the description of the used development environment, this chapter presents detailed information about the Slimcompiler, which compiles Java class files into the new slim file format. Furthermore, this chapter describes the callback functions to the server and it also presents detailed information about modified opcodes, describes basic blocks and the Exceptionhandling.

Chapter 6 gives an overview of the benchmarks used for the evaluation. Furthermore, the test environment is described and finally the evaluation of the taken measurements of the SlimVM approach in terms of memory space savings, class file size reduction and effect on performance is presented.

Chapter 7 presents the conclusions drawn from this master's thesis and discusses future work for the SlimVM approach.

Chapter 2

Background

This chapter gives an introduction and overview of the overall functionality of JamVM, an insight into the Java class file format, and describes the bytecode format examplied by a sample program. Furthermore, this chapter describes the Byte Code Engineering Library BCEL and the Java Native Interface.

2.1 JamVM

JamVM [Lou04] is an open source Java Virtual Machine (JVM) written in C with a small memory footprint. The size of the stripped executable on PowerPC is approximately 200KB and on Intel approximately 180KB. Unlike other lightweight JVMs (e.g. KVM [Sun99b]) it supports the full Java specification (version 2 - blue book) [LY99] including object finalization, the Java Native Interface (JNI) and the Reflection API. JamVM uses the GNU Classpath Java class library [GNU99] and uses a mark-sweep garbage collector (GC), which can be run either synchronously or asynchronously within its own thread.

As every other JVM, JamVM is a stack-based machine. For each thread (each new Java program to be executed by the same virtual machine) a JVM stack, which stores frames, is created. The use of a frame is to store data and partial results, as well as return values for methods, dispatch exceptions and to perform dynamic linking. Each time a method is invoked, a new frame (Figure 2.1) is created and pushed onto the frame stack. When its method invocation completes whether normal or abrupt (it thows an uncaught exception) the frame is destroyed. Only the frame of the current executing method is active at any point.

As mentioned in Listing 2.1, each frame consists of an operand stack (ostack), an array of local variables (lvars), also called the local variable table, as well as a reference to the current method which references the current class and therefore the current constant pool. Furthermore, a frame holds the current program counter (last_pc) and a pointer to the previous frame (prev).

Listing 2.1: Internal representation of a frame in JamVM

```
1 typedef struct frame {
2    CodePntr last_pc;
3    uintptr_t *lvars;
4    uintptr_t *ostack;
5    MethodBlock *mb;
6    struct frame *prev;
7 } Frame;
```

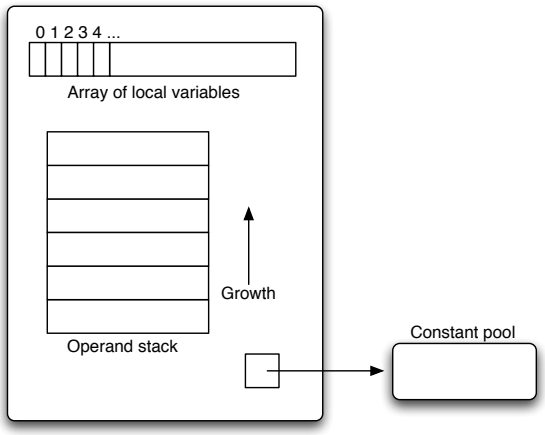

Figure 2.1: *A frame in a Java Virtual Machine [Hag01], created for every invoked method, which consists of the local variable table, the operand stack and a reference to the current constant pool.*

The local variable table is used to store the values of the local variables but is also used to receive the parameters of the method. The parameters are stored first, beginning at index 0. The array size for local variables depends on the number and size of local variables and formal method parameters and is determined at compile time. As listed in Listing 2.2, in JamVM, this information is stored in the variable *max_locals*.

The operand stack is a last-in-first-out (LIFO) stack, whose size is also determined at compile time. In JamVM, as listed in Listing 2.2, this value is stored in the variable *max_stack*. Certain bytecode instructions push values onto the operand stack whereas other instructions take operands from the operand stack, manipulate them, and push the result back onto the operand stack. Furthermore, the operand stack is used to receive return values from methods.

As listed in Listing 2.2, a *MethodBlock* in JamVM holds a lot more information like the method's name, type and signature. The type u2 represents an unsigned two-byte quantity. The *method_table_index* in line 20 indexes into the virtual method table and is used to support the dynamic binding of Java. The *Exception table* in line 18 is used to lookup, and to jump to the correct exception handling whenever an exception is arised. Line 15 shows the pointer to the actual bytecode.

Listing 2.2: Internal representation of a method in JamVM

```
typedef struct methodblock {
    Class *class;
    char *name;
    char *type;
    char *signature;
    u2 access_flags;
    u2 max_stack;
    u2 max_locals;
    u2 args_count;
    u2 throw_table_size;
    u2 exception_table_size;
    u2 line_no_table_size;
    int native_extra_arg;
    void *native_invoker;
    void *code;
    int code_size;
    u2 *throw_table;
    ExceptionTableEntry *exception_table;
    LineNoTableEntry *line_no_table;
    int method_table_index;
    MethodAnnotationData *annotations;
} MethodBlock;
```

Figure 2.2 gives an overview of the overall architecture of JamVM in which the *Interpreter* is the main core of the system because the *Interpreter* executes the actual bytecode. The bytecode is a stream of instructions where an instruction may invoke a new method, therefore a new frame is pushed onto the operand stack. The *Interpreter* calls the *Linker* which checks whether the class of the method has been loaded so far or not. If the class of the method is not loaded at this time, it is requested from the *Class Loader* and the *Linker* links the class in the Java virtual machine.

The *Class Loader* is responsible for the dynamic loading and thereby creating the internal data structures for classes. The *Linker* is responsible for linking a class or interface, which involves verifying and preparing that class or interface, its direct superclass, its direct superinterfaces, and in case of an array type, its element type.

An instruction may also create a new object which is then stored on the heap. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The GC is responsible for the storage of objects on the heap. An instruction may also call a native function via the *Java Native Interface*.

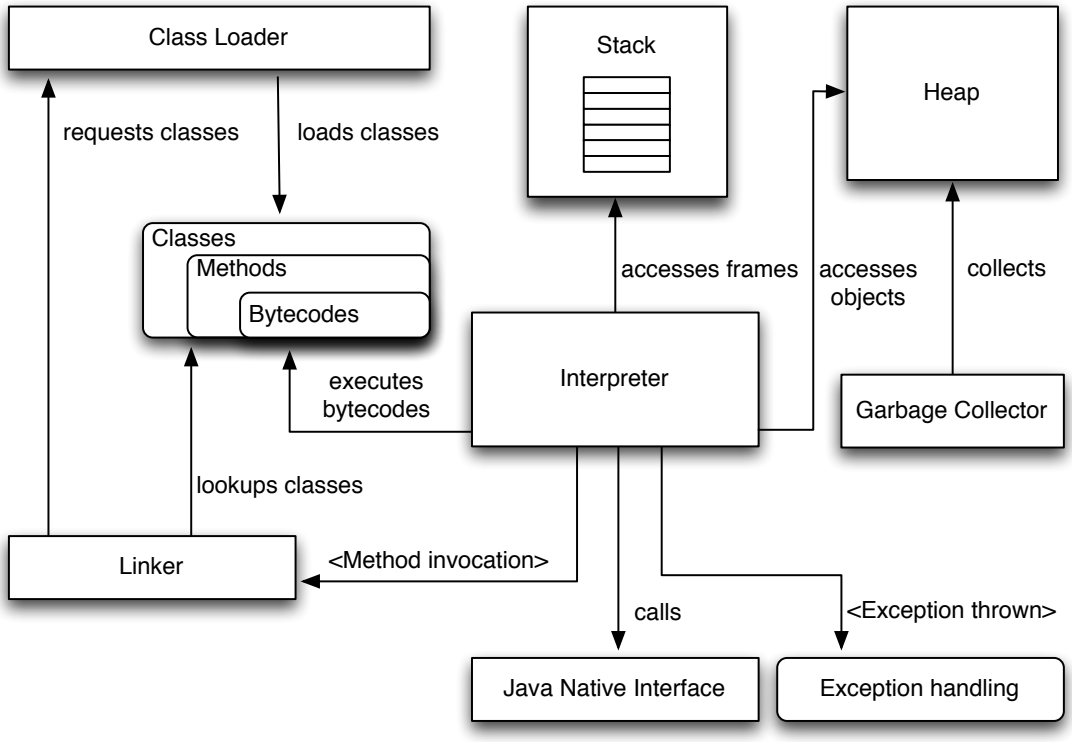

Figure 2.2: *Architecture of JamVM: The main core of the system is the interpreter because it interprets the actual bytecode instructions. A method invocation instruction causes the Linker to lookup the requested class. If the class has not been loaded so far it is requested from the Class Loader. Finally the method is invoked and a new frame is pushed onto the Stack.*

2.2 Class File Format

A Java compiler such as as javac translates a Java program into a hardware and operating system independent binary format, typically (but not necessarily) stored in a file, known as the *Class File Format*. Figure 2.3 illustrates the compilation of a Java source code file to a Java class file. The format of class files is described in more detail in the *Java Virtual Machine Specification* [LY99], and in [MD97]. Every class file defines the representation of a class or interface and consists of a stream of 8-bit bytes where multibyte data items are stored in big-endian order.

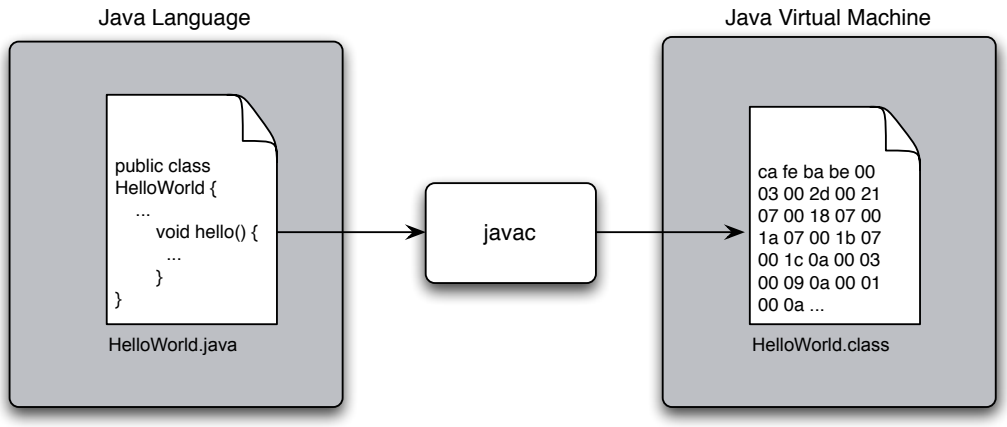

Figure 2.3: *A Java source file is compiled by a compiler such as javac into the machine independent Java class file format.*

Listing 2.3 shows the *Class File Format* in detail where the types u1, u2, and u4 represent an unsigned one-, two-, or four-byte quantity. The * stands either for an internal used structure for *attributes*, *methods*, *fields*, or the *constantpool*.

Listing 2.3: Java Class File Format

```
1  Class File {
2      u4 magic:
3      u2 minor_version;
4      u2 major_version;
5      u2 constant_pool_count;
6      * constant_pool[constant_pool_count-1];
7      u2 access_flags;
8      u2 this_class;
9      u2 super_class;
10     u2 interfaces_count;
11     u2 interfaces[interfaces_count];
12     u2 fields_count;
13     * fields[fields_count];
14     u2 methods_count;
15     * methods[methods_count];
16     u2 attributes_count;
17     * attributes[attributes_count];
18 }
```

The Java Class File of Listing 2.3 can be grouped into seven basic sections which are listed and described below:

Header

The header of the class file starts with a magic number (0xCAFEBABE) to identify a Java class file and is followed by the minor_version and major_version of the class file format being used.

Constant pool
The constant_pool_count indicates the number of entries in the following constant pool table. Since constants in the constant pool table start at index 1, this count is one greater than the actual number of entries. constant_pool is the actual constant pool table, which is an array of variable size constants containing strings, numbers and references to methods and classes. The runtime constant pool is described in more detail in Section 2.4.

Access rights
This section holds the access_flags of the class encoded by a bitmask and indicates whether a class is public, abstract, final, etc. Furthermore, this_class holds the name of the current class and super_class holds the name of the super class.

Implemented interfaces
The interfaces_count indicates the number of direct implemented superinterfaces of the class which are stored in the following interfaces table.

Fields
The fields_count indicates the number of fields in the fields table including class fields and instance fields, declared by this class or interface type.

Methods
The methods_count indicates the number of methods which gives a complete description of a method in this class or interface. If the method is native or abstract, the instructions for the JVM implementing the method are also supplied.

Class attributes
The attributes_count indicates the number of attributes which are stored in the following attributes table.

Since the Java virtual machine is designed to dynamically resolve symbolic references to methods, classes and fields at run-time, all these references are encoded as string constants stored in the constant pool. In fact, the constant pool contains the largest portion of an average class file, approximately 60% [AP98], whereas the bytecode instructions themselves just make up 12% of an average class file.

2.3 Byte Code Format

The *Byte Code Format* [LY99] is the specification of instructions that the Java virtual machine executes. A Java virtual machine instruction is a one-byte opcode followed by zero of more operands. Many instructions not have any operands and consist only of an opcode. Not all of the possible 256 instructions are used. The instruction set currently consists of 212 instructions, 44 are marked as reserved and may be used for optimizations within the VM. The Java virtual machine instruction set can be grouped as follows:

Load and Store Instructions

These instructions are used to load values from the local variables (e.g. iload) onto the operand stack of a Java virtual machine frame and vice versa (e.g. istore). Another important opcode of these category is ldc, which loads a constant from the constant pool onto the operand stack.

Arithmetic Instructions

These instructions compute a result of two values on the operand stack and push the result back on the oprand stack. Two main kinds of arithmetic instructions can be differed: instructions operating on integer values and instructions operating on floating-point values. For example, iadd, adds two integers. Since there is no direct support for integer arithmetic for the types byte, short, char and boolean, these types are handled as integers in the JVM.

Type Conversion Instructions

These instructions allow the conversion between Java virtual machine numeric types. E.g. i2l, is a numeric conversion of an int to a long.

Object Creation and Manipulation

Even though class instances and arrays are objects, the JVM distinguishes between these two types and uses different instructions for creation and manipulation. E.g. new creates a new class instance whereas a new array is created with one of the following instructions: newarray, anewarray, multianewarray. To access fields of objects the Java virtual machine uses getfield, putfield and respectively getstatic, putstatic for static fields.

Operand Stack Management Instructions

Within the Java virtual machine there are some instructions that manipulate the operand stack directly. E.g. pop, which pops a value of the operand stack or dup, which duplicates the value on top of the operand stack.

Control Transfer Instructions

Branch instructions like ifgt or goto could cause the Java virtual machine to continue execution with an instruction other than the one following the branch instruction.

Method Invocation and Return Instructions

There are four instructions that invoke a method within the Java virtual machine: invokevirtual, which invokes an instance method of an object; invokeinterface, which invokes a method that is implemented by an interface, invokespecial, which invokes an instance method that requires special handling; invokestatic, which invokes a static method.

The return instructions of methods are distinguished by their type. E.g. ireturn, returns an integer.

Throwing Exceptions

An exception can be thrown either programmatically with the athrow instruction or by Java virtual machine if an abnormal condition is detected.

Except the lookupswitch and tableswitch instruction, which are used to implement switch statements, all instructions of the Java virtual machine have a fixed length. But since the number of cases in a switch statement may vary, the size of these instructions may also vary.

2.4 The Runtime Constant Pool

The constant pool [LY99] is a table of constants in each class that contains values, ranging from numeric constants known at compile time, to method, field and class references that must be resolved at run time. In order to keep the bytecode short, typically all constants are referenced by the bytecode using an index into the constant pool. Table 2.1 shows all the possible types for constants within the constant pool:

Table 2.1: Java Reference Types for Constant Pool Entries

Constant Type
CONSTANT_Empty
CONSTANT_Utf8
CONSTANT_Integer
CONSTANT_Float
CONSTANT_Long
CONSTANT_Double
CONSTANT_Class
CONSTANT_String
CONSTANT_Fieldref
CONSTANT_Methodref
CONSTANT_InterfaceMethodref
CONSTANT_NameAndType

At some point during every running Java program, methods, fields, classes, etc. must be resolved. The process of finding and replacing the symbolic reference with a direct reference is called resolution. Resolution in Java follows a simple schemata. The actual Java bytecode indexes into the constant pool. The constant at this index is either a primitive type like a CONSTANT_Utf8 which holds for example a field-, method- or classname, or it refers to other entries in the constant pool which holds further information in order to resolve the correct field, method, class, etc. The connection between the possible reference types is illustrated in Figure 2.4.

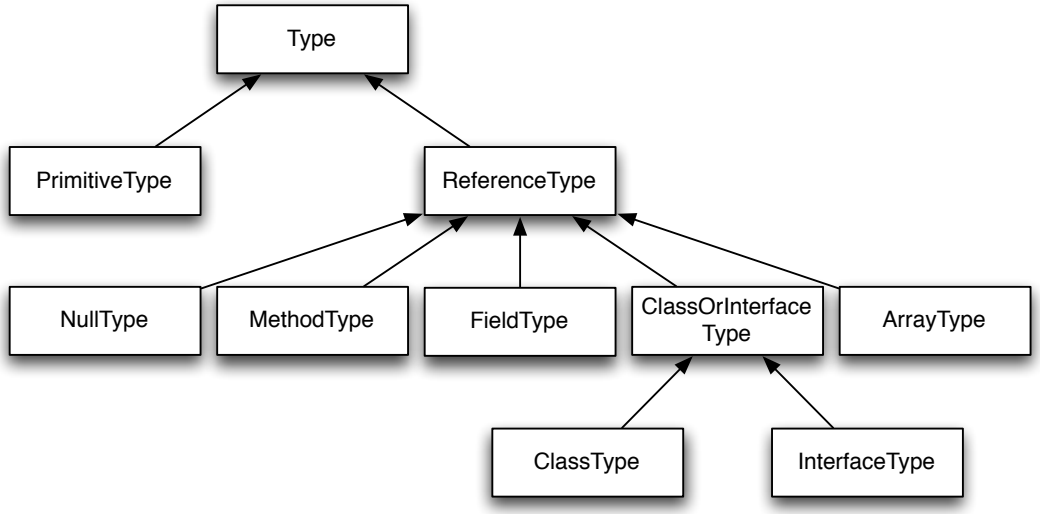

Figure 2.4: Java Reference Types.

The left side of Figure 2.5 shows the class file format of the file Helloworld.class. The right upper box extracts some parts of the constant pool of this class and the box below examines the instructions of the translated program of Listing 2.4.

Listing 2.4: Java source code for a Hello World program

```
1 public class HelloWorld{
2   public static void main(String args[]){
3     System.out.println("Hello, world");
4   }
5 }
```

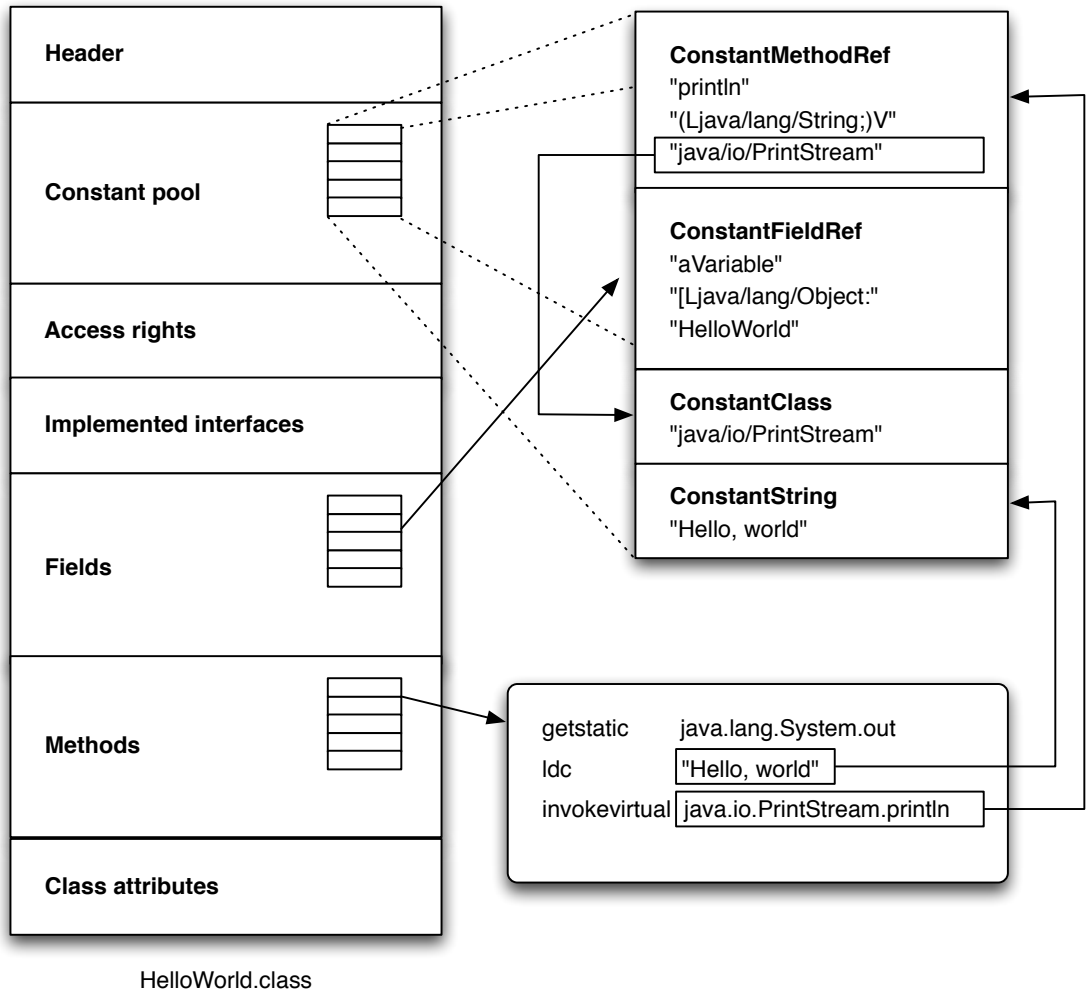

Figure 2.5: Java class file format [Dah01].

The first instruction loads the contents of the field out of the class java.lang.System, which is an instance of the class java.io.PrintStream, onto the operand stack. Using the ldc instruction, a reference of the constant string "Hello, world" is pushed on the operand stack. The invokevirtual instruction finally invokes the instance method println which takes both values as parameters. As indicated, the instruction invokevirtual refers to a MethodRef constant, which contains information about the method such as the name, the signature and to which class the method belongs. In fact, the MethodRef constant itself just refers to other entries in the constant pool holding the real data, e.g. it refers to a ConstantClass which contains a symbolic reference to the class java.io.PrintStream.

2.5 Byte Code Engineering Library

The *Byte Code Engineering Library* (BCEL) [Dah01] is a tool which provides a simple API for decompressing, modifying, and recomposing binary Java class files. BCEL exposes all of the binary components and data structures declared in the JVM specification [LY99] as objects. These objects may be used to modify and even to generate new bytecode. Figure 2.6 illustrates the UML diagramm for the BCEL API.

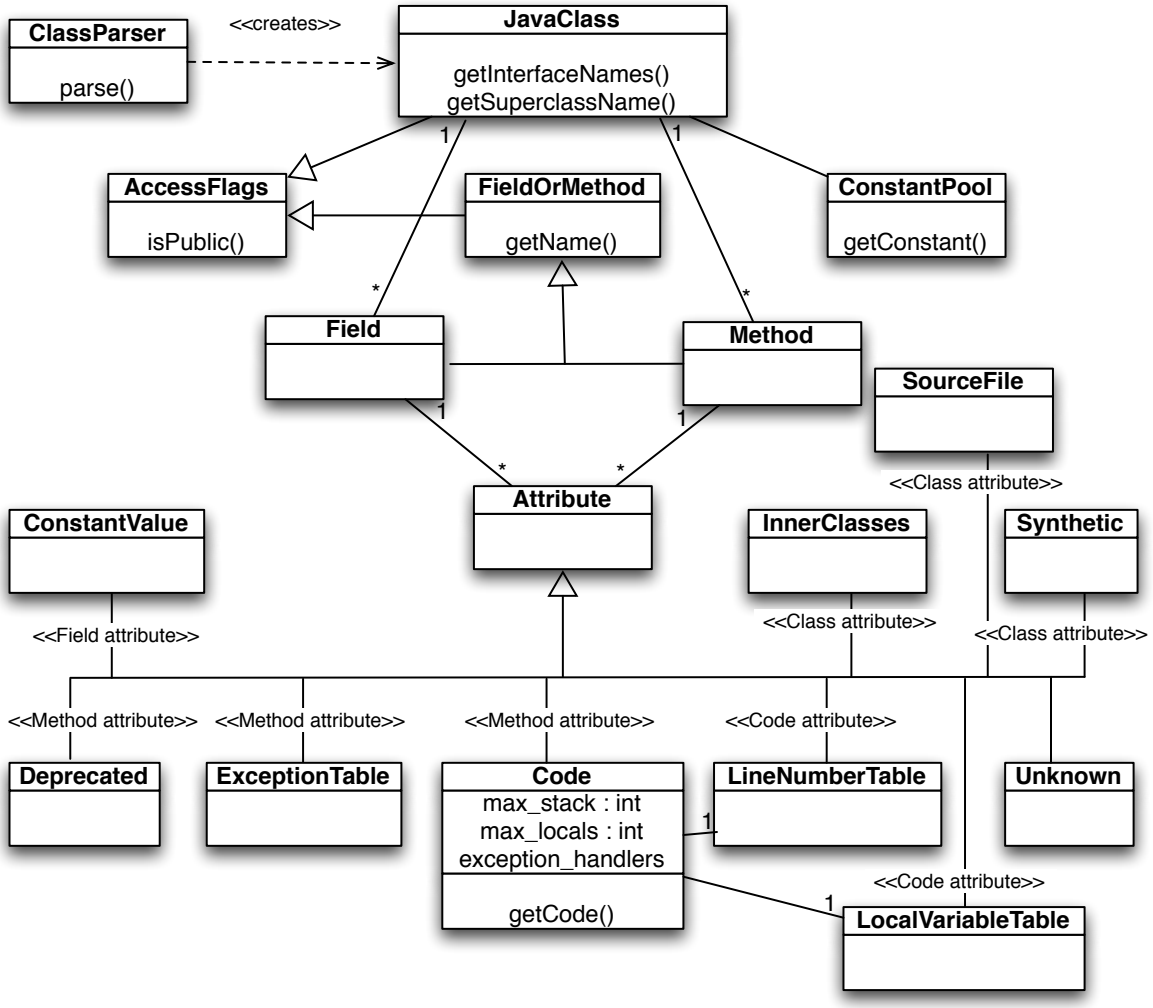

Figure 2.6: UML diagramm for the BCEL API [Dah01].

The BCEL API mainly consists of three parts:

1. A package of classes which expose all elements of class files like fields, methods, etc. This package may be used to read and write class files from or to a file but is not intended for bytecode modifications.

2. A package to dynamically generate or modify JavaClass objects. For example, this package is used to insert analysis code or to strip unnecessary information from class files. Furthermore, it is used to implement the code generator back-end of a Java compiler.

3. Various utilities like a converter from class files to the Jasmin assembly language [MD97], a class file viewer and a tool to convert class files into HTML.

The BCEL API in its general purpose is a tool for bytecode engineering which helps developers implementing analysis tools or bytecode transformations. The BCEL library has been proved to be useful in several diverse applications and is not restricted to a specific kind of application area.

2.6 Java Class Library

The *Java Class Library* [Wik08] is a set of pre-written classes which can be dynamically loaded by any Java application during runtime. Like other standard code libraries, the *Java Class Library* provides the programmer a well known set of functions to perform common tasks, such as maintaining lists for example. In addition, class libraries provide an abstract interface to tasks that normally depend heavily on the hardware of the operating system (OS), such as file and network access. Currently, only some different Java class libraries are available, like those from Sun or IBM's Virtual Machines or OpenJDK.

JamVM uses *GNU Classpath* [GNU99] and does not work with any other class library. GNU classpath is part of the Free Software Foundation's GNU project and creates a free software implementation of the standard class library for the Java programming language. The aim of the GNU classpath project was to give computer users the possibility to use Java programs without giving up the freedoms which the FSM (free software movement) works to secure. GNU classpath consists of 7258 classes with a total size of 14.89 MB.

2.7 Java Native Interface

The *Java Native Interface* (JNI) [Lia99] is a native programming interface that allows Java code that runs inside the Java virtual machine to interoperate with applications (hardware and operating system specific programs) and libraries written in other languages, such as C, C++, and assembly. Figure 2.7 illustrates the role of the Java Native Interface:

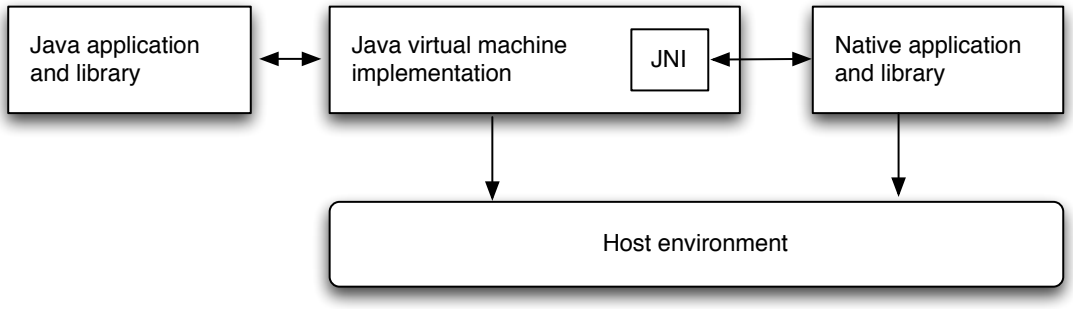

Figure 2.7: Role of the Java Native Interface [Lia99].

The following examples illustrate when Java native methods are needed:

- Platform-dependent features that are needed by the application and that are not supported in the standard Java class library.
- Already existing libraries written in another language should be accessible to the Java code.
- A small portion of time critical code is implemented in a lower-level language such as assembly.

The JNI is a powerful feature and designed to handle situations where Java applications need to be combined with native code such as when the standard Java class library does not support platform-specific features or program library. A lot of the standard Java library classes depend on the JNI to provide functionality such as I/O file reading. Since JNI is included in the standard library, this performance- and platform-sensitive API allows all applications written in Java to access this functionality in a safe and platform-independent manner. By programming through the JNI, native methods can be used to:

- Create, inspect, and update Java objects.
- Call Java methods.
- Catch and throw exceptions.
- Load classes and obtain class information.
- Perform runtime type checking.

In JamVM, all native functions are implemented in a separate .c file. Every time the JVM invokes the function, it passes the JNIEnv pointer, a jobject pointer, and any Java arguments declared by the Java method. A JNI function to get the length of a string may look like this:

Listing 2.5: JNI method to get the length of string

```
1 jsize Jam_GetStringLength(JNIEnv *env, jstring string) {
2     return getStringLen((Object*)string);
3 }
```

The env pointer is a structure which includes all of the functions necessary to interact with the Java virtual machine and to work with Java objects. JNI functions can convert native arrays to/from Java array, are able to convert native strings to/from Java strings, are able to instantiate objects, throw exceptions, etc. Basically, everything Java code can do can be done by using the JNIEnv.

2.8 Code Example

The example program in Listing 2.6 calculates and prints the factorial of the static integer n. Even though this is not the best way of calculating the factorial of a number, this example serves its demonstrating purpose. The println function may raise an Exception and therefore the critical area of code is encapsulated in a try-catch block.

Listing 2.6: Java source code for a factorial program

```
1  public class Faculty {
2    public static int n = 4;
3
4    public static void fac(){
5      int f = 1;
6      int i = n;
7      while(i > 0){
8        f *= i;
9        i--;
10     }
11     try{
12       System.out.println(f);
13     }
14     catch(Exception e){ System.err.println(e); }
15   }
16
17   public static void main(String[] argv){
18     fac();
19   }
20 }
```

Java bytecode differs between two constructors, a static constructor <clinit> for initializing all static fields, which is called only once for each class, and an instance constructor <init> which is called separately for every instance of a class. Since the factorial example only has a static field, only the static constructor is printed below, because the instance constructor is negligible for the example. In Listing 2.7 the instruction iconst_4 loads the value 4 on the stack and putstatic stores this value into a static field.

Listing 2.7: Bytecode of the static constructor <clinit> of a Factorial program

```
0: iconst_4
1: putstatic     #10; /Field n:I
4: return
```

The static function fac() called by the main function of this program is printed in Listing 2.8. Value 1, loaded by the instruction iconst_1 and value 4 of the before initialized static field are stored in slots 0 and 1 in the local variable table.

The instruction goto jumps to offset 16 which is for the ease of reading displayed as an absolute address even though branch instructions are actually relative. The value 4 of slot 1 is loaded onto the stack again. Because 4 is greater than 0, the ifgt instruction jumps to address 9 which loads the two values of slot 0 and 1 which are then mulitplied by imul. The result is pushed back on the stack and stored in slot 0 of the local variables. iinc decrements the value stored in slot 1 and iload_1 loads this value on the stack again. This is done until the value of slot 1 is 0. This part of the bytecode represents the while-loop of the Java programming language.

The instructions between address 20 and 27 are used to print the result. If no exception is thrown the instruction goto jumps to offset 38 and the method returns. If an exception is thrown, a lookup in the Exception table is necessary where the target indicates the new jump address, in this case 30 where the exception handler starts.

Listing 2.8: Bytecode of the method fac() of a Factorial program

```
 0: iconst_1
 1: istore_0
 2: getstatic  #10; /Field n:I
 5: istore_1
 6: goto 16
 9: iload_0
10: iload_1
11: imul
12: istore_0
13: iinc 1, -1
16: iload_1
17: ifgt 9
20: getstatic  #20; /Field java/lang/System.out:Ljava/io/PrintStream;
23: iload_0
24: invokevirtual #26; /Method java/io/PrintStream.println:(I)V
27: goto 38
30: astore_2
31: getstatic  #32; /Field java/lang/System.err:Ljava/io/PrintStream;
34: aload_2
35: invokevirtual #35; /Method java/io/PrintStream.println:
                                                (Ljava/lang/Object;)V
38: return

Exception table:
from to target type
20   27 30     Class java/lang/Exception
```

Chapter 3

Related Work

Many researchers have identified the importance of code size reduction, dead code elimination and bytecode compression for ensuring Java's success on embedded systems. Therefore this chapter references the latest research proposals in this field to give an overview of the state of the art, including the first developed approach SlimVM.

3.1 JAR Format

The standard means of packaging Java class files for distribution and storage is Sun's JAR (Java ARchive) [Sun99a] file format. The JAR format is based on the well known zip [PKW89] compression mechanism to aggregate many Java class files as a single unit. In fact, it is possible to extract members of JAR files using a zip decoder. JAR also allows other data types to be stored among with the class files.

JAR reduces the size of Java class files by almost 50% and can be executed by any Java virtual machine. Since Java 1.5.0 the compression utility Pack200 [Sun08c] is included which was designed for compressing JARs and works efficiently on Java class files. Using Pack200 compression the size of a JAR file can be reduced by about 60%.

3.2 JAZZ: An Efficient Compressed Format for Java Archive Files

Bradley et al. present a format called Jazz [BHV98]. Similar to JAR, Jazz [BHV98] bundles a number of class files together and compresses them. Bradley et al. eliminate redundant constantpool entries and combine the constantpools of all compressed classes to one unified constantpool within the Jazz archive. In this way, any constant string will only appear once, no matter how many classes make use of it. Also method names, signatures, integer constants only appear once. Huffman [NG96] codes are used for all indices into the unified constantpool as it is an optimal method of assigning variable length codes to symbols.

To achieve a good compression, Bradley et al. sort all strings in all the classes, concatenate them, compress them using zip and store them in one part of the Jazz file.

Furthermore the length of strings has to be stored. To do this in an efficient way, all strings are sorted by length and delta coding (a delta code is the difference between one value and the next) is used to encode the length of the strings. As the strings are sorted by length, the values will normally be small and therefore they will need fewer bits to store. In fact, the presented Jazz format reduces the data to 25% of the original size.

3.3 Compressing Java Class Files

Pugh presents the wire-code format [Pug99], a custom compressed format for collections of Java class files. A compressor transforms the JAR file into the wire format using three main approaches to create smaller files that contain the same information as a JAR file:

1. Transmission of redundant information by sharing information across class files. As classnames are encoded as a package name and a simple class name, Pugh changed the representation so that all classes from the same package share the package name and therefore it only has to be stored once (e.g. java.lang only occurs once). Also classes from different packages can share the same simple class name.

2. Types of methods and fields are not encoded as strings. Instead, Pugh encodes method types as an array of classes containing the return and argument types. A field type is just a class and primitive types and array types are encoded as special class references which are converted back when decompressed.

3. Elimination of generic attributes. Pugh sets additional flags in the access flags instead, which indicate whether specific attributes apply.

A decompressor has to be present on the target machine which needs 36 KB in JAR format. As the wire format is a sequential format, all of the class files must be decompressed in sequence. Once decompressed they can be transformed back to a convential JAR file or separate classfiles. The wire format is typically 50% to 80% smaller than the original JAR file using the compression algorithm gzip [Deu96].

3.4 Compact Java Binaries for Embedded Systems

Rayside et al. present an effective solution for the conflicting requirements of size reduction and execution performance called compact Java Binaries [RMH99]. Regular class files can be converted to the new format and back easily and quickly, but executing the new format requires either a customized class loader or a slightly modified virtual machine.

Rayside et al. identified the constantpool and the code attribute of the class file as the two most important factors to reduce the size of the file. In order to reduce the size of the constantpool Rayside et al. explicitly represent the package tree structure and the hierarchical organization of types in Java. Due to this alteration it is possible to replace string references to types with indices to the explicit representation. Furthermore Rayside et al. separate the opcodes from the operands and apply three different techniques in order to reduce the size of the code attribute:

- The first approach uses the Huffman algorithm in order to reduce the number of bits required to represent the most frequent opcodes.
- The second approach is based on the observation that some opcodes often appear in pairs. Rayside et al. associate each pair of opcodes with a unique prefix code using the Huffman algorithm again.
- The third approach is an attempt to exploit existing code patterns generated by Java compilers such as javac. A Markov state model helps to represent each opcode as a unique state. The first state can then represent a sequence of opcodes and a sequence of state transitions. This representation helps Rayside et al. to identify distinct groups of opcodes that a likely to be followed by opcodes of another group. Again, the Huffman algorithm is used to determine the prefix codes that represent the possible transitions between all the states.

Rayside et al. modified the constantpool and the code attribute of class files in a way that their evaluation show a typical size reduction of 25% for class files and 50% for JAR files.

3.5 Java Bytecode Compression for Low-End Embedded Systems

Clausen et al. present in their work *Java Bytecode Compression for Low-End Embedded Systems* [CSCM00] that factorization of common Java bytecode instructions can reduce the memory footprint, on average, to 85% of its original size, with a minimal time penalty. Although Clausen et al. preserve Java compatibility, their solution requires some modifications in a JVM used in a low-end embedded system.

As the standard Java class file format includes unnecessary information not needed to be present in a low-end embedded system, it is common to use its own internal space-efficient representation. Clausen et al. use the JavaCard [Sun08a] technology. A Java program is transferred to JavaCard systems in units of packages. A package is converted into a single CAP (converted applet) file and an export file which describes the package interface. Other export files which describe other packages that are used by the classes in the package allow all the name information to be stored in export files, so that two byte tokens are the only representation of names in the CAP file. Once the CAP file is transferred onto the JavaCard device, it can be converted into whatever internal representation is used for execution.

Furthermore, Clausen et al. use code factorization to eliminate code redundancy. Their proposal is to extend the virtual machine to read new instruction definitions from the CAP file. These *macro instructions* replace common instruction sequences in the code and can be stored with little memory overhead in the run-time system. Conceptually, recurring sequences of operations are abstracted by factorizing them into single units. Each bytecode instruction sequence is called a *pattern*. With this approach, the only limitation of new instructions is the number of unused instructions in the standart instruction set. Factorization for a given program is a two step process:

- First, repetitive instructions have to be identified as patterns. All possible combinations of instruction sequences occurring in the program are generated in order to find the best set of patterns to factorize the program where identical sequences are treated as a single *occurrence group*. The virtual machine must keep track of the current package in order to reference constants correctly as in the CAP file each package has its own constantpool. Furthermore, Clausen et al. consider the instructions *tableswitch, lookupswitch, jsr,* and *ret* as unfactorizable and therefore they are removed from a pattern by splitting the pattern into two new patterns, and splitting the occurence group accordingly. Similarly, whenever an outgoing branch leaves a pattern, the branching instruction is removed from the pattern, creating two new patterns.

- Secondly, the bytecode is factorized and new instructions are generated on the fly. Clausen et al. generate the macros greedily by selecting the occurrence group that gives the most savings and keep on doing that either they run out of unused instruction codes or occurrence groups that save space. Each occurrence is replaced by a macro instruction and any other occurrence that contained the replaced occurrence needs to be updated in order to reflect this change.

3.6 Generation of Fast Interpreters for Huffmann Compressed Bytecode

Latendresse and Feeley [LF03] use canonical Huffman codes to create an instruction set for a customized VM and present an implementation of that machine that directly executes this compact code. Their approach creates either new instructions in order to replace a sequence of instructions, or a basic instruction with a new format for the operands.

First, they build a dictionary of (possibly overlapping) repetitive sequences. Secondly, they create a dictionary of formats to encode all basic instructions using as few bits as possible. Third, a greedy algorithm considers the maximum space saving and selects either a sequence of instructions or a new format until no space gain can be obtained. Furthermore, the greedy algorithm considers the opcode length, the new formats and the space of the decoder. They encode the opcodes using static frequencies of the opcodes from a sample of programs to generate canonical Huffman codes with variable lengths. Canonical Huffmann codes are similar to the original bottom up method, with the difference that the numerical values of the codes of a given length form a consecutive sequence. Therefore, they have a compact representation of the bijection between the codes and the encoded object.

In order to increase the speed for decoding, Latendresse and Feeley use a table which contains branching addresses at which either decoding continues or the virtual instruction is decoded. They distinguish between three different look-ups:

1. The opcode is recognized and a direct jump to the implementation of the virtual instruction is done.

2. The opcode is not recognized but its length is known. In this case the length of the opcode is used to compute its index by equation and a jump to the implementation of the virtual instruction is done.

3. The opcode is not recognized and its length is unknown. In this case, the following bits are used to continue decoding using another look-up. Therefore the decoder has a tree structure where every interior node is case 3.

Case 1 and 2 are leaf nodes. Each type 3 node requires a vector of addresses of its own, whereas type 2 nodes share the same vector. They use the basic parameters, the (static or dynamic) frequencies of the opcodes and a given space constraint to generate the fastest decoder. A branch and bound algorithm is used which searches from the fastest to the slowest decoders and stops when the space constraints are met.

Latendresse and Feeley present a solution where the compression factors highly depend on the original bytecode and the training sample, but typically vary from 30% to 60%.

3.7 A Java Bytecode Optimizer using side code analysis

Clausen presents Cream [Cla97], a Java bytecode optimizer which performs dead-code elimination, loop-invariant removal as well as side-effect analysis. Cream consists of nine phases where the first two phases are interprocedural, while the remaining phases are performed for one method at a time:

Class graph construction
In order to make it possible to analyze interface and virtual methods a class graph is constructed which contains all classes and superclasses that may be used by the program. Classes which have instances allocated during runtime are marked as such.

Side-effect analysis
Side-effect analysis for virtual methods is done before the main analysis is started. Methods not reached by the *main()* method are only analyzed on demand.

Control-flow graph construction
Instructions are divided into blocks, where the control flow may change. The control flow is then represented as edges between these blocks. Furthermore, instructions that may throw an exception also end a block. During this phase Clausen also separates the control flow from the order of the instructions to make it easier to move instructions around.

Stack depth inference
In order to find out how high the stack is at any given point, Clausen traverses the control-flow graph and identifies stack variables by their offset from the bottom of the stack, not by their position relative to the stack pointer.

Use-def analysis
Clausen builds use-def chains for local variables and stack variables using a standard

analysis with kill-sets and gen-sets with the only difference that stack variables are always considered killed when used.

Cycle detection and dominance graph construction
Clausen uses a cycle detector in order to build a dominance graph before he does the loop invariant elimination.

Dead code marking
Clausen uses a dead-code marker which considers all impure instructions, return instructions and instructions that may change the control flow to be live. Instructions not live or used or used to construct anything live will be eliminated in the transformation phase.

Loop-invariant detection
Instructions which are invariant during a loop are detected in this phase. If the loop itself is found side-effect free, this may also include field accesses and invocations of methods without side-effects.

Code transformations
In this final step, the possible transformations are performed. Clausen mentioned that the challenge is to keep the stack consistent. For example, when the result of a method invocation is no longer used, because the use turned out to be dead code, but the result must still be popped off the stack.

Clausen presents an optimizer for Java bytecode using inter-procedural side-effect analysis with a possible performance improvement of up to 25%.

3.8 Practical Extraction Techniques for Java

In order to reduce the size of an application for embedded Systems, Tip et al. present a number of extraction techniques [TSL+02] such as the removal of unreachable methods and redundant fields, inlining of method calls and the transformation of the class hierarchy. They implement their techniques in *Jax*, an application extractor in Java. Jax reads the Java class files of the original application and constructs an in-memory representation of those files. Only classes that contain the application's entry points and classes that are directly or indirectly referenced from those classes are loaded by *Jax*.

Removal of Redundant Class File Attributes
Unnecessary class file attributes like the local variable name tables and line number tables are discarded during loading.

Removal of Unreachable Methods
During the execution of a program only a subset of methods in the loaded classes is required by an application. There are many reasons why unreachable methods arise. A well known example is that applications often use class-libraries that are developed somewhere else. In some cases methods cannot be removed for syntactic

reasons, in these cases *Jax* still removes the body of the method and replace it with a `return` statement.

Redundant Field Elimination
Fields only accessed from unreachable methods can be removed from the application. Furthermore, as Tip et al. presented in 1998 [ST98], fields only written to, but never read, can also be removed from the application because those fields cannot affect the program's behavior.

Class Hierarchy Transformations
Jax eliminates entire classes and merges adjacent classes in the hierarchy.

Name Compression
Classes, methods and fields are currently renamed in *Jax*. Tip and Sweeney use more ambitious naming schemes and use the same name for methods with different signatures.

Performance Optimizations
Even though the primary goal was to reduce archive size, Tip and Sweeney optimized the performance by inlining non-overriden methods in cases where this does not increase the size of the application. Moreover, they replaced an *invokevirtual* with an *invokespecial* where a virtual dispatch has only one potential target.

Constant Pool Compression
After the transformations described above have been performed, the in-memory representation of *Jax* creates new class files. These class files are written out again and a new constantpool is created from scratch, where only classes, methods, fields and constants actually referred are added.

Tip and Sweeney reduce the size of class file archives, on average, to 37.5% of their original size.

3.9 Generic Adaptive Syntax-Directed Compression for Mobile Code

Stork et al. present an approach [SHF00] for mobile code compression operating on abstract syntax trees (AST) which can be parameterized by abstract grammars (AG). This makes their approach applicable for any source language without further language-specific modifications. Stork et al. use novel statical approaches to compress the AST of a program. They achieve a more compact encoding by using domain knowledge about the underlying language since the AST is composed according to a given AG.

First, an AST is poduced from the source code during the *parsing process*. In order to store and transport ASTs they need to be serialized. Stork et al. do this by generating the ASTs pre-order representation. The AG can be effective on the actual tree representation and much information in the pre-order encoding is redundant. In order to use as few bits

as possible for encoding, Stork et al. use an arithmetic coder [WNC87] as it is the best means to encode a number of choices if each alternative has a certain probability.

In the prototype implementation Stork et al. use the Barat [BS98] framework for parsing and after decompressing the binary file, the prototype interfaces directly with GCC as code-generating backend. With this approach Stork et al. reduce the size of Java classes (packages) by 5-50%.

3.10 Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments

Saougkas et al. [SMBZ07] customize agglomerative clustering (AC) toward the identification of parameterized and nonparameterized patterns in order to reduce the size of Java bytecode. This hierarchical pattern discovery technique helps them to discover patterns of arbitrary length containing a variable number of wildcards. Saougkas et al. present the advantages and limitations of the afore mentioned technique using MIDP [Sun08b], a standard Java environment that supports application development for mobile devices. Their overall compression process consists of four steps:

1. They segment the Java bytecode into *basic blocks*.
2. They use a pattern discovery technique to identify possible patterns in the basic blocks of the bytecode.
3. The resulted patterns are collected and possible combinations of these patterns are examined. Furthermore, the bytecode size reduction for each pattern combination is calculated.
4. Finally, the pattern combination which gives the best bytecode size reduction is selected and used for the generation of the compressed code.

Nonparameterized Pattern Discovery
The nonparameterized pattern discovery is in fact a simple string search problem. The first step of the code compression process identifies a set of basic blocks of the Java bytecode. Then, the pattern discovery tries to find common substrings that are repeated in these sequences and the subset giving the best bytecode reduction is stored in a dictionary.

Parameterized Pattern Discovery
The parameterized discovery technique is an extension of the nonparameterized technique and starts from the point where the different collections of substrings of sequences have already been identified.

Saougkas et al. use AC, an hierarchical approach relying on the bottom-up generation of a treelike structure of clusters. Starting with a set of clusters (leaf nodes) the algorithm searches at each next step the current set of clusters to identify the two most similar ones

and merges them into a new cluster. The algorithm terminates when no pair of nodes is allowed to be further merged.

Saougkas et al. developed a pattern discovery method that reduces the bytecode size of a MIDP package, on average, by about 15% to 30%.

3.11 Slim Binaries

Franz and Kistler present slim binaries [FK97], a compact platform-independet program representation, which is designed to be translated into binary code by an optimized JIT (just-in-time) compiler.

Their implementation generates high quality native code on-the-fly which is fast enough that it can compete with the loading of the compiled code from an usual binary. Slim binaries not only solve the problem of compatibility between different architectures, they also allow to fine-tune the object code towards the specific processor and operating system version that it will run on.

Slim binaries encode the abstract syntax tree of a program and can be verified easily, because the code can be restricted to valid syntax trees of the programming language. Thus, expensive bytecode verication can be avoided.

Based on the fact that different parts of a program are often similar to each other (e.g. procedures in typical programs are often called again and again with almost identical parameter lists), these similarities are exploited by using an *predictive compression* algorithm which allows the encoding of recurrent subexpressions in a program space efficiently. Due to the structured representation which can include information needed for optimizations, the compilation overhead is negligible, and the generated binary code is as efficient as that generated by an ordinary compiler. This approach can reduce the size of a complete application by a factor of three.

3.12 Code Compression

Ernst et al. present a compressed executable called BRISC [EEF+97] (Byte coded RISC), an interpretable VM code with about the same size as an non-interpretable gzipped x86 program.

They assume transmission and memory as the two most important criteria for the execution of a program. Unlike slim binaries which compresses full executables, they compress only code segments.

The compressor starts by adding the base instruction set, which currently consists of 224 instruction patterns, to a dictionary of frequently occurred instruction patterns. Furthermore the compressor scans the input program over and over, to find and to add useful instructions to the dictionary and to calculate their program size reduction and their cost in decompressor memory. They calculate the program size reduction by calculating the reduction in compressed program bytes that would occur if the candidate instruction pattern were added to the dictionary minus the number of bytes needed to represent the

instruction pattern in the dictionary. For the code generation or interpretation the BRISC decompressor uses a table of native instruction sequences.

To keep track of the candidate instructions the compressor maintains a heap. After each pass over the input program, the compressor adds the best candidates to the dictionary and removes them from the heap. After that, the compressor modifies the input program to reflect the newly available instruction patterns. The compressor maintains a table that maps each base instruction pattern to a list of all input program instructions that matches that pattern to avoid undue overhead. Similarly, the compressor also maintains a hash table of all previously generated instruction patterns so that generating of already existing candidates can be avoided.

Once a dictionary is created, the dictionary followed by the modified input program which has been compressed during the construction of the dictionary are written to the BRISC file. As the results show, BRISC is a good mobile program representation choice which is competitive with gzip in code size.

3.13 SlimVM: Optimistic Partial Program Loading for Connected Embedded Java Virtual Machines

Wagner et al. present a zero footprint paradigm for connected embedded devices [Wag07] [WGF08], where all code resides on a remote network host and is provided only on-demand. First, all application and library class files are loaded by the Slimanalyzer, a tool based on BCEL [Dah01], which analyzes all application and library code and generates an optimistic subset of class files which are likely needed by the client. During this phase, Wagner et al. identify all required information of the application and library, extracts them and compresses this subset.

Pre-linked bytecode is transferred to the mobile device and if their heuristic fails and code marked as unlikely executed is needed during execution, the client sends a code reload request to the server before continuing execution. They consider three different levels of granularity:

Class Level
When the client requests a class file, only the basic information like static and instance variable size, static and instance constructor and the remaining strings in the new constantpool are transferred.

Method Level
They present two different approaches for their method level, an optimistic and a pessimistic one. In the **pessimistic approach**, every method referenced by an *invoke* is shipped to the mobile device during basic class loading. In the **optimistic approach** all methods except of *init* and *clinit* have to be requested during runtime. This is effective for classes only requested because of a static field.

Basic-Block Level
Wagner et al. change the bytecode representation by removing basic blocks from

methods. Their analysis tries to identify those parts of methods that are going to be executed. Code less likely to be executed, like exception code, remains on the host network and is only shipped on demand. In order to do so, all instructions of a basic block are replaced by a placeholder instruction which references to the corresponding basic block with a numeric identifier. Each removed basic block is replaced by an instruction called *block*. Whenever such an instruction is reached during execution, the corresponding basic block will be requested from the server.

In order to reduce the size of the constantpool, they change the representation of references from strings to numbers. The same is done for names of methods, fields, classes and types. As a result, they drop the whole constantpool with the exception of strings which cannot be replaced, like "Hello World!" which should appear on the screen. These strings are now kept in a separate string stream.

For their prototype *SlimVM* implementation they use the *K virtual machine*[Sun99b] on the client and in order to execute the new format, they modified these VM. Opcodes, formerly followed by an operand indexing in the constantpool are now followed by the class or method identifier number and therefore no further lookup into the constantpool is needed. Furthermore, they use a lookup table for each class which stores the transferred status and the corresponding offset for each method. Each field instruction can be resolved to an absolute offset within the corresponding object and no further time-consuming lookup is necessary.

The results of Wagner et al. show that code-size reduction, pre-linking and on-demand provision reduce the size of Java applications for connected embedded mobile devices by up to 95%. Furthermore, their approach decreases the overall memory consumption of the VM by as much as 75%.

Chapter 4

Design

This chapter presents the principle approach and the design of SlimVM. It describes the customized and optimized Java virtual machine on the client, includes details of the client-server architecture and the communication between them. Furthermore, this chapter includes use case diagrams for the SlimVM approach and a data flow diagram.

4.1 The Principle Approach

The principle approach of SlimVM is the idea that a Java virtual machine can be separated into two main parts. One part, which is in charge of analyzing and pre-linking the whole program and the other part, which has just the function to execute the modified bytecode instructions.

As illustrated in Figure 4.1, firstly a Java program has to be compiled into java class files which are then read and analyzed by the Byte Code Engineering Library BCEL. BCEL maps all class and method names against numerical identifiers in order to keep the size of the new slim file format small and in order to support pre-linking. Pre-linking means that the resolution of all bytecode instructions which refer into the runtime constant pool is done on the server prior to execution on the client and replaced by the appropriate numerical identifier in case of a method invocation. That means that no constant pool has to be transferred in the new slim file format, but that also means that a lot of instructions have to be modified which are described in detail in Section 5.5.1.

The slimcompiler compiles every Java class file into the new slim file format which does not include any actual bytecodes at all. The bytecodes are kept separately in form of basic blocks as it is the smallest granularity of Java bytecode.

The two parts of the Java virtual machine are separated on a server and on a client. All program code remains on the server and is requested by the Java virtual machine on the client during runtime. Since class file information and bytecodes are seperated it is possible to transfer just the information necessarily needed for execution of the program. Whenever a new class is needed, it is requested from the server and whenever a method of that class is invoked, only the bytecode needed for execution is requested from the server.

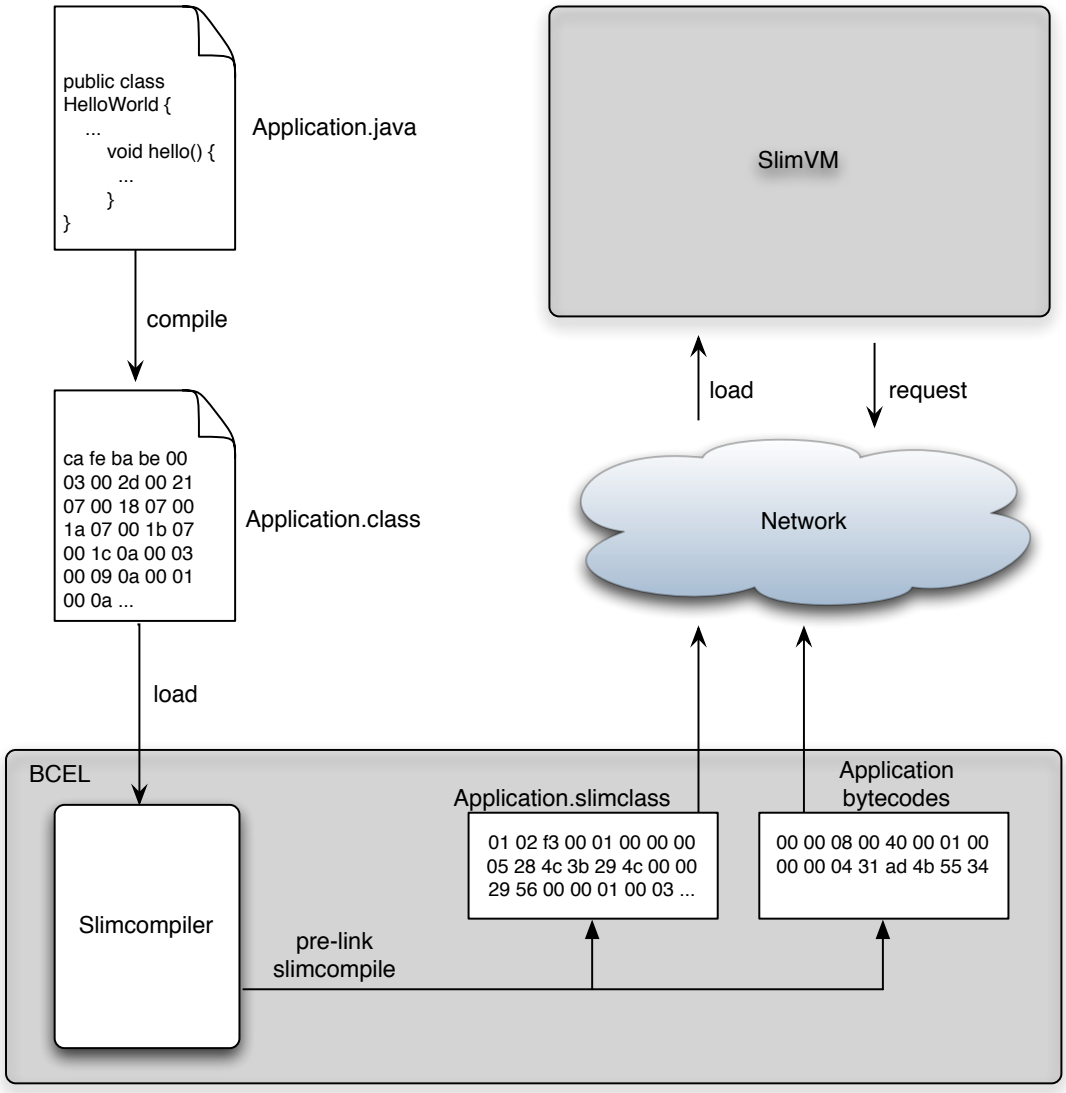

Figure 4.1: *All java source files have to be compiled into java class files which are then read by BCEL. The slimcompiler compiles every class file into a new slim file format which does not contain any actual bytecode instructions at all. All bytecode instructions are kept separately and can be requested individually in form of basic blocks by the Java virtual machine on the client.*

In order to execute a program it is not necessarily needed to have the whole program code to be present on the mobile device. Figure 4.2 illustrates the execution flow of Listing 4.1 and shows that Code Block 1 and Code Block 2 are never executed by that program.

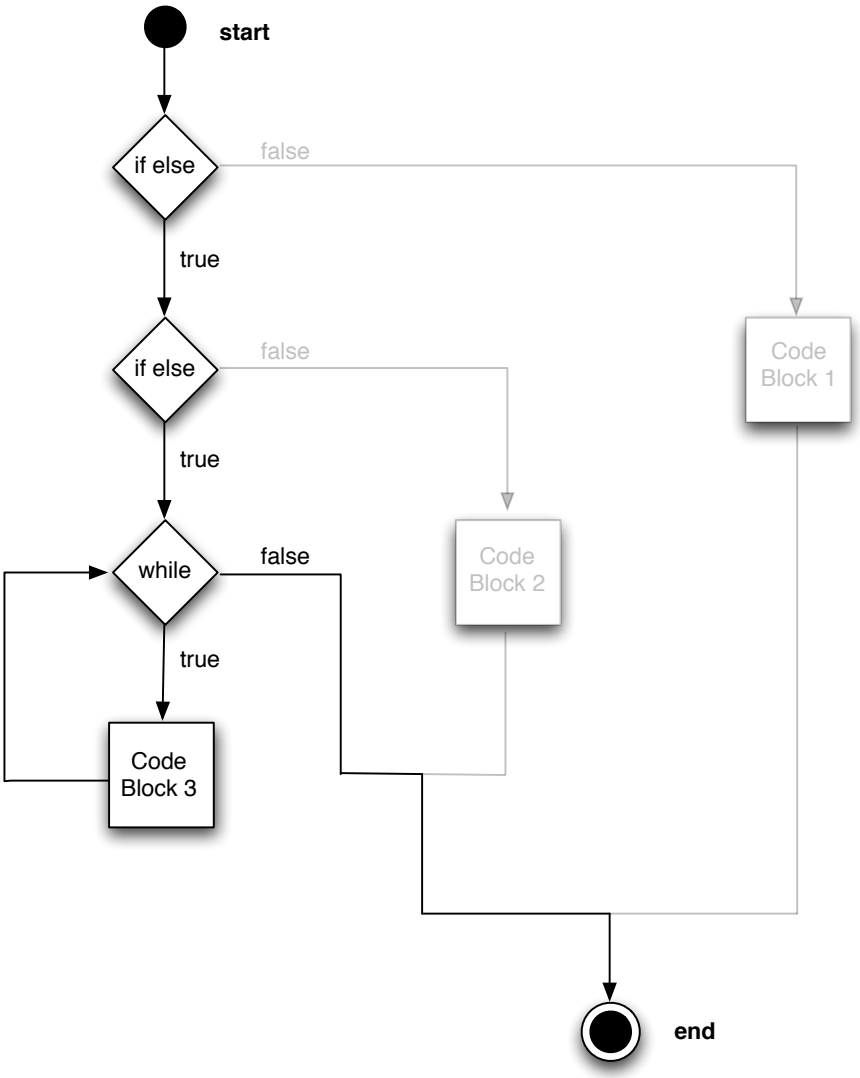

Figure 4.2: *This Figure shows the execution flow of a method. As illustrated, only a subset of code is actually needed to execute a method.* Code Block 1 *and* Code Block 2 *are not needed for execution of that method.*

Listing 4.1: Example to show the execution flow of a method

```
1  public void foo(){
2    int i = 0;
3    if(i < 10){
4      if(i < 5){
5        while(i < 5){
6            Code Block 1
7        }
8      }else{
9          Code Block 2
10     }
11   }else{
12       Code Block 3
13   }
14 }
```

In order to keep the memory footprint small on the mobile device, it is crucial to send only bytecodes necessarily needed for execution a program. For example, Code Block 1 and Code Block 2 could be enormous in size including hundreds of hundreds of instructions but are never executed and therefore unnecessary for the execution of that program.

4.2 System Architecture

The main idea for the development of SlimVM is the reduction of the code and memory footprint of a Java virtual machine on an persistently connected embedded mobile device. All code (including library and application code) of a Java program is analyzed prior the execution on the mobile device. During this step, all classes referenced by the main() method are read and analyzed recursively. That means, that every Java class file which is invoked by an instruction in the currently analyzed file is also going to be read and analyzed. The information collected for field, method, and class resolution is evaluated and compiled into a slim class file format developed for this approach.

As bandwidth is always a bottleneck for persistently connected embedded mobile devices, this new invented class file format holds information needed for resolution within the runtime constant pool not based on strings, as the original class file format does. Instead, it uses a new way for field, method, and class resolution which is based on numeric identifiers. Furthermore, all information not necessarily needed for execution of a class file is dropped, including the complete Attribute section. That means, that no bytecode instructions are present in the new class file. In order to support this new invented slim class file format, the actual bytecode of a class file is manipulated and resides on the network host until the mobile client requests parts of it.

To start execution of a Java program on the client, only the slim class file that includes the main() method, has to be present on the mobile device. Once execution of the program has started, the client requests classes in form of the new slim file format and bytecode in form of pre-linked basic blocks, as this is the smallest granularity of Java bytecode. Hence, only code indispensably needed for execution of a Java program is shipped to the

mobile device and therefore the code and memory footprint of the Java virtual machine is reduced.

4.2.1 Client-Server Architecture

Figure 4.3 illustrates the overall client-server architecture of the SlimVM approach. On the server-side, we use the *Byte Code Engineering Library* BCEL to analyze all application and library class files referenced by the Java program to be executed. Since BCEL offers, amongst others, a package to dynamically generate JavaClass objects, we use this package to strip unnecessary information of a class file and to compile the analyzed information gathered, into the new slim class file format. Immediately after the compilation process is finished, and all classes are written into the new slim file format, the server, which we also programmed using BCEL is started.

In order to develop SlimVM on the client-side, we modified and customized the Java virtual machine *JamVM*, as it is a virtual machine whith a small memory footprint. Since JamVM uses the GNU classpath and does not work with any other Java library path, we have to analyze these library class files on the server.

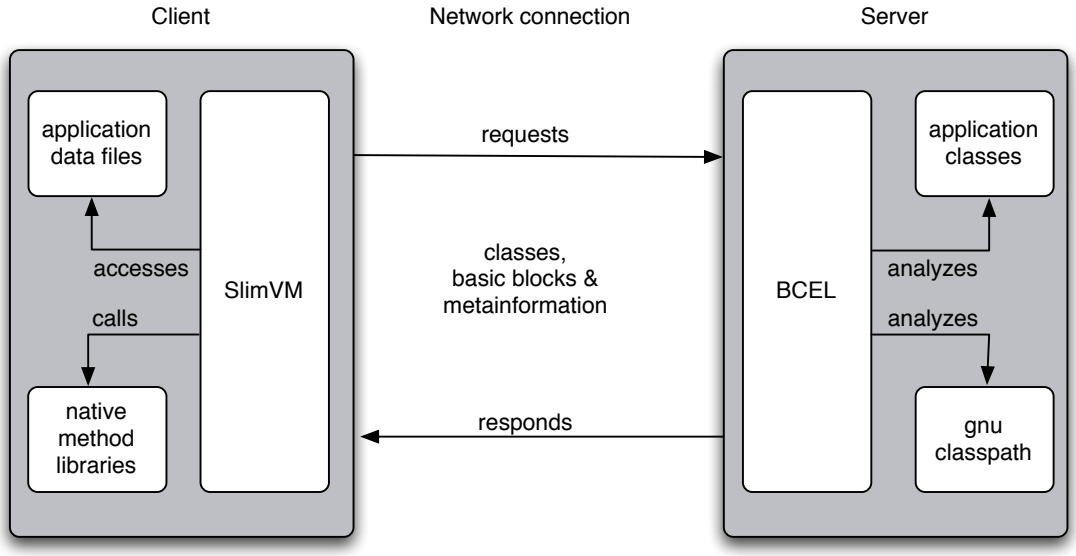

Figure 4.3: Client-server architecture of SlimVM.

Once execution on the client is started, classes in the new format and bytecode in form of pre-linked basic blocks are requested from the server. To do so, the server offers a number of callback functions, either to request missing data, or to transfer metainformation needed for execution on the client. This is necessary to support the *Java Native Interface* on the client. For example, assume that a native method (all native method libraries are present on the client) is called by the Java program. As I replaced all method names by a distinct numeric identifier, the corresponding native method can't be called, because

native methods are called by their name. Therefore, the client needs to callback to the server to request the corresponding method name, before the native method can be called and execution of the program continuous.

4.2.2 SlimVM

To support *on demand code loading*, interpret modified instructions, and to load classes in the new format, we customized the Java virtual machine *JamVM*. Even though it is still a stack based machine, we carried out a lot of changes to make it feasible for the new approach. Figure 4.4 illustrates the overall architecture of the modified Java virtual machine *SlimVM*.

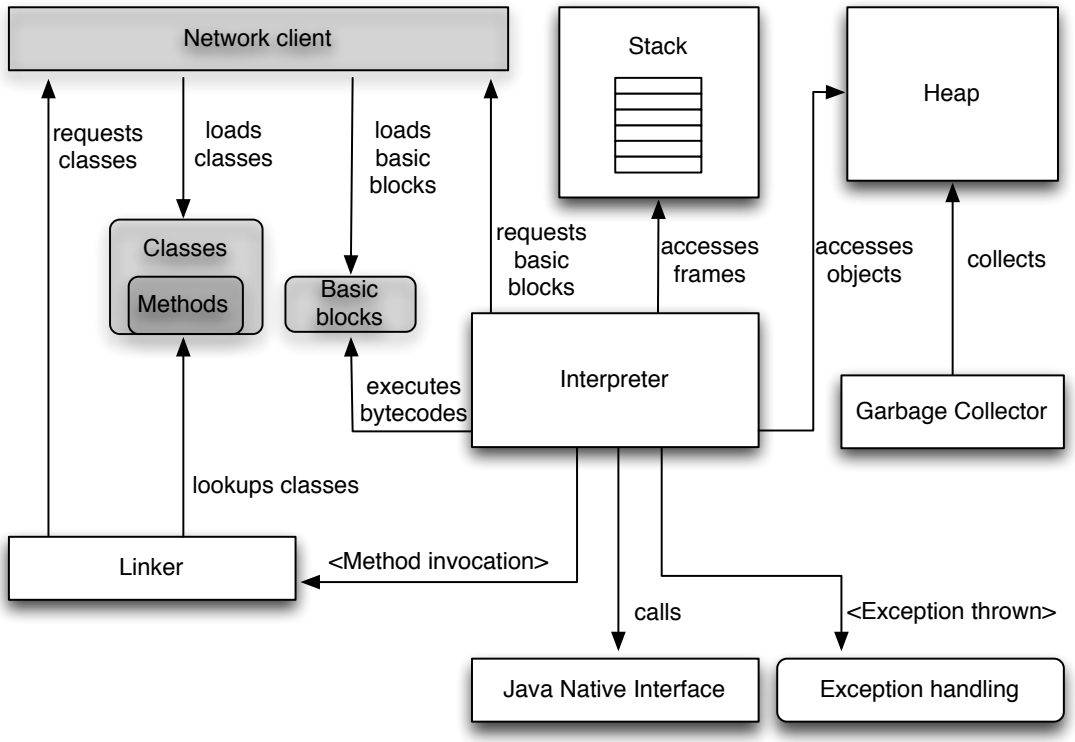

Figure 4.4: Architecture of SlimVM.

Whenever execution of a new application is started, at first, SlimVM tries to establish a TCP connection to the server. Since no bytecode is present on the mobile device at startup, SlimVM shuts down if connecting to the server fails, because then no class information and no bytecode instructions can be requested, and therefore no Java program can be executed. If the connection is finally established, SlimVM starts its initialization process. Some basic classes like java.lang.Object, java.lang.Class or java.lang.String are transferred in the new slim file format to the client. The initialization process includes initialization

of native methods, DLLs, *Java Native Interface*, etc. and the initialization of the heap size.

After the virtual machine is initialized, the static method main() of the program to be executed is invoked and the first basic block of this method is requested from the server. The interpreter starts executing the instructions of the first basic block. Whenever a new method is invoked, the *Linker* looks up the class of the method. If the class has not been transferred to the mobile device so far, it is requested from the server and loaded through a customized class loader. This modification of the class loader is necessary, because class files are transferred in the new slim file format. When the class is loaded, the method is finally invoked, therefore a new frame is popped onto the frame stack, and the first basic block of this method, is also requested from the server. If the interpreter reaches a branch instruction or the end of the basic block, simply the next or the corresponding basic block, in case of a branch instruction, is requested.

Since all Java class files are analyzed on the server prior of execution on the client, I developed a new slim file format which makes the representation of classes, methods, etc. as string constants dispensable. Thus, every string constant is replaced by an numerical identifier which makes the usage of the constant pool redundant.

As illustrated in figure 4.5, every class consists of several methods and every method contains of basic blocks which hold the actual bytecode instructions.

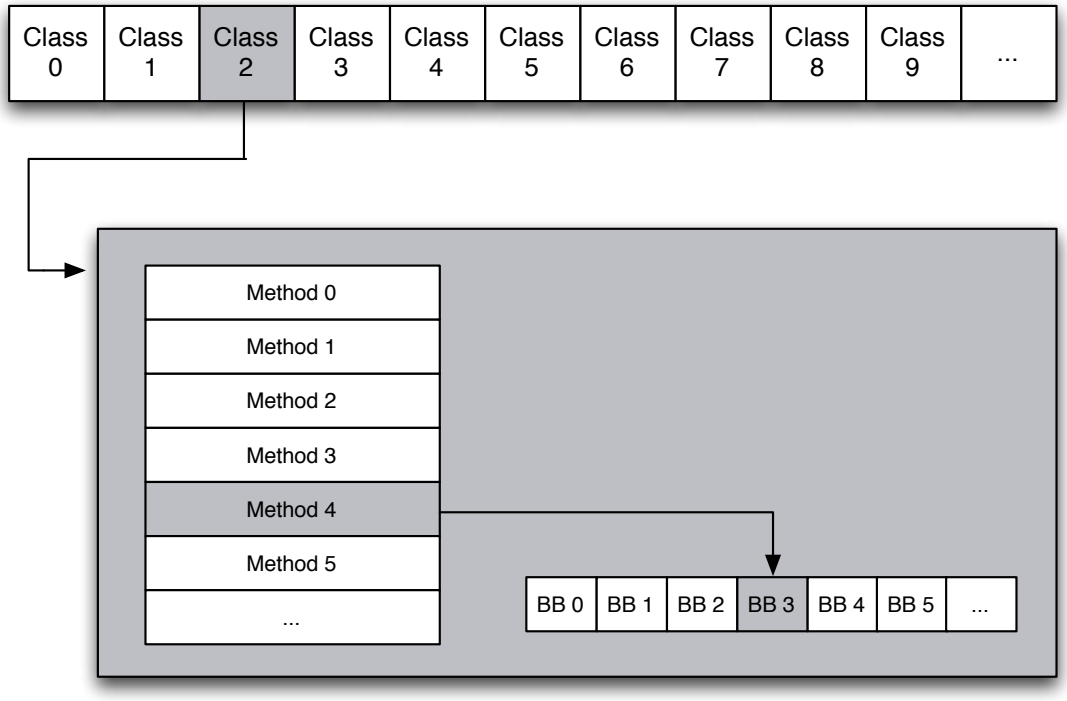

Figure 4.5: Interaction of classes, methods, and basic blocks.

Some basic blocks are needed for execution of a Java program and others don't. For example, if an if statement is executed, the else statement can't be executed. Therefore only bytecode instrcutions for the if branch are relevant and needed for execution on the client. Thus, only bytecode instructions necessarily needed for execution of the Java program need to be transferred to the mobile device.

To avoid transferring the same basic block over and over again when the method is called repeatedly, every transferred basic block is stored on the client in a format illustrated in Listing 4.2.

Listing 4.2: Internal representation of a basic block in SlimVM

```
1 typedef struct slimbasicblock {
2    int basicblockid;
3    int codesize;
4    unsigned char *code;
5 }SlimBasicBlock;
```

When the invoked method is a *native method*, the method is programmed in native code (in case of SlimVM in C), then no bytecode instructions for this method are available and can't be requested from the server. But since the name of the method was replaced by the numerical identifier and native methods are called by name, the client needs to call back to the server in order to request the native name of this method.

Usually, every Java class file contains of a Code Attribute which normally holds the actual bytecode instructions. But as I use basic blocks as the smallest granularity of bytecode, I dropped the whole Attribute section of the class file and therefore, every method holds basic blocks instead of the whole bytecode instructions as illustrated in Listing 4.3.

Listing 4.3: Internal representation of a method in SlimVM

```
1 typedef struct slimmethodblock {
2   Class *class;
3   int methodid;
4   int isstatic;
5   char *type;
6   int innerid;
7   u2 basic_block_count;
8   SlimBasicBlock **basic_blocks;
9   int currentbbid;
10  int isnative;
11  void *native_invoker;
12  char *nativename;
13  int native_extra_arg;
14  u2 max_stack;
15  u2 max_locals;
16  u2 args_count;
17  void *code;   (necessary for JNI methods)
18 } SlimMethodBlock;
```

Furthermore, a *SlimMethodBlock* holds information like the method's numerical identifier, whether the method is static or not. And some information needed to support

Reflection in Java, like the *innerid*, which is used, for example, to create the correct constructor object. Line 10 indicates, whether a method is native or not. If the method is native, then this method doesn't hold any basic blocks. Instead, the nativename of line 12 is requested from the server in order to call the correct native method. Either the native_invoker points to the correct native method if available, otherwise the JNI has to be called and therefore line 17 points to the corresponding code.

4.3 Client-Server Communication

Client and server are connected over a TCP connection. The Server offers seven different callback functions which are described in more detail in Section 5.4.1.

Figure 4.6 illustrates how class information can be requested from the server. First, the client calls the GetClassData() function with the numerical class identifier as parameter. The server then looks up that class in BCEL and sends the length of the bytecode to be sent back to the client. The client receives the length and sends back OK to the server. Then the server sends the requested bytecode.

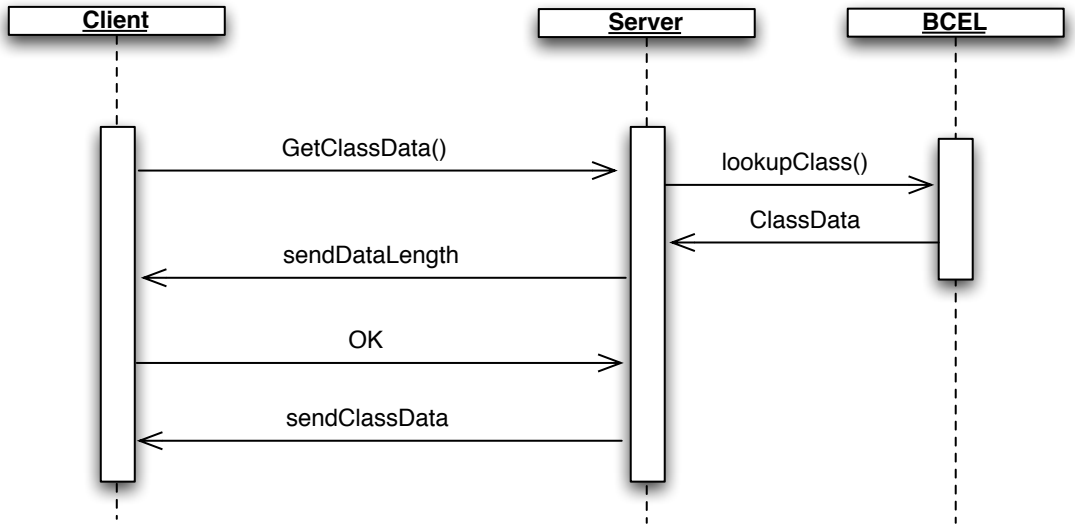

Figure 4.6: Sequence Diagram for getClassData().

Figure 4.7 illustrates how a basic block can be requested from the server. The client calls the function getBasicBlock given the numerical class identifier, the numerical method identifier and the number of the requested basic block. With this information the the server looks up the basic block in BCEL and sends the length of the data to be sent back to the client. The client answers with OK and the server sends the requested basic block.

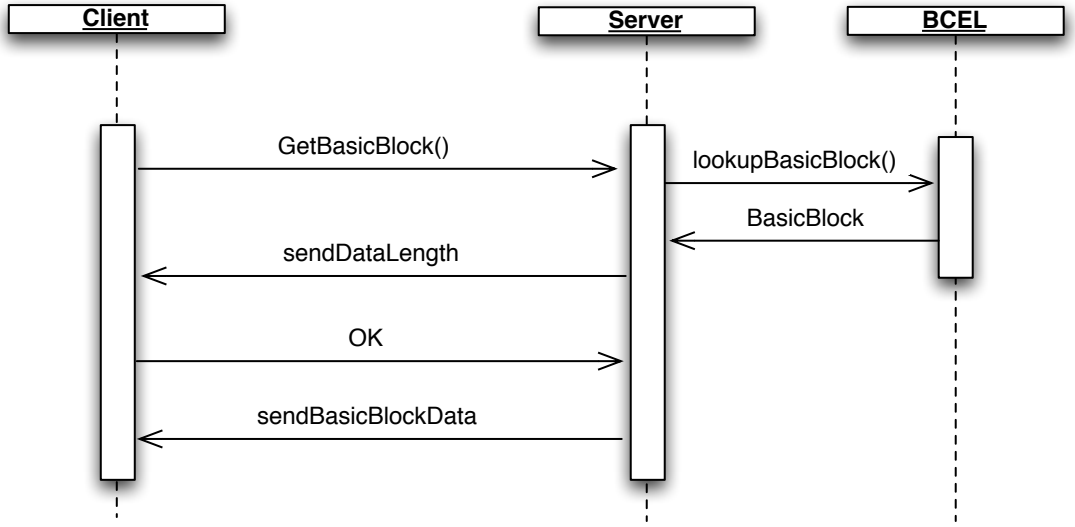

Figure 4.7: Sequence Diagram for getBasicBlock().

Figure 4.8 is meant to give an overview of how metainformation can be requested form the server. As illustrated the client wants to request the Number (identifier) of a basic block which catches an thrown exception. The client calls the function GetExceptionBasicBlockId given the numerical identifier of the class, the identifier of the method and the number of the basic block including the offset where the exception occured. The server looks up the instruction in BCEL, after some more look ups the corresponding catching basic block is found and the requested number of the basic block is sent back to the client.

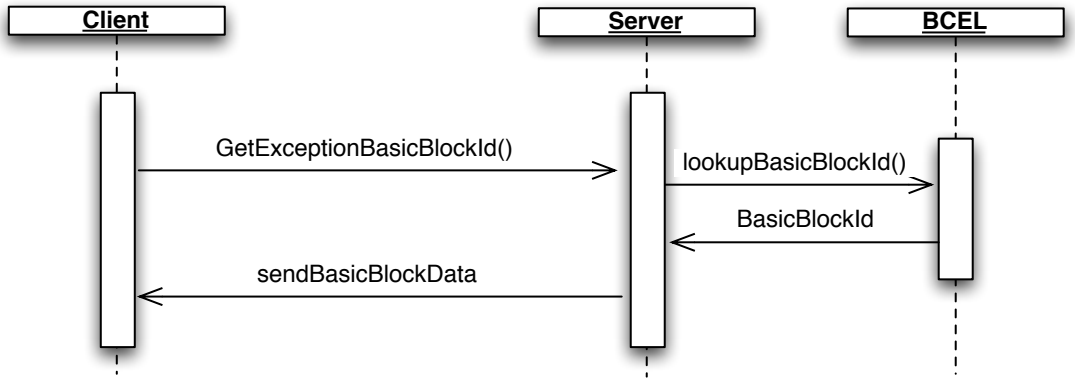

Figure 4.8: Sequence Diagram for getExceptionBBid().

4.4 Use Case of SlimVM

Figure 4.9 illustrates a typical *Use case* for executing a Java program in SlimVM. The Actor starts executing the Java program on the mobile device by starting SlimVM with the slim class file including the main() method as parameter.

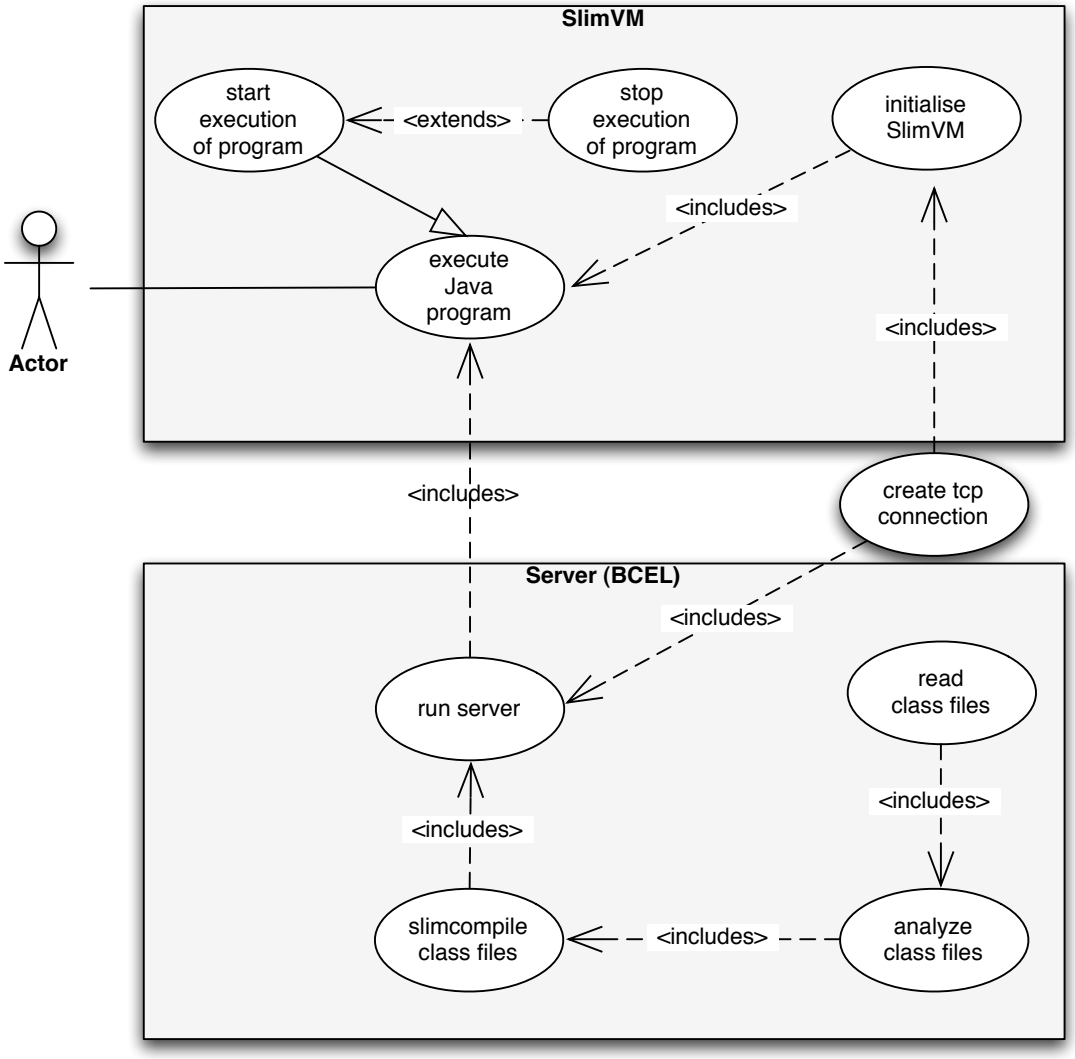

Figure 4.9: Use case of execute Java program.

Table 4.1 shows the gathered description of the *Use case* execute Java program.

Table 4.1: Use case description of execute Java program

Use case name	**Execute Java program**
Goal	Execute a Java program in SlimVM
Summary	An actor starts a Java program to be executed by the modified Java virtual machine SlimVM
Actors	User, SlimVM on Client, Server (BCEL)
Preconditions	- all class files need to be present on the server - TCP socket needs to be ready for connection
Triggers	Actor starts executing the Java program by starting SlimVM with the slim class file including the main() method as parameter
Basic course of events	- class files are read into BCEL - class files are analyzed in BCEL - class files are compiled into the new slim file format - execution of the program starts - a tcp connection is established - SlimVM is initialised - classes, basic blocks, metainformation are requested from the server in order to execute program - execution of program finished
Alternative paths	
Postconditions	
Notes	
Author and date	Christoph Kerschbaumer, November 21, 2008

4.5 Data Flow in SlimVM

Figure 4.10 illustrates a reduced Data Flow Diagram in SlimVM. SlimVM starts executing a Java program and whenever a class needs to be requested from the server, SlimVM calls back to the server and requests that needed class. The same scenario for basic blocks of a Java program. After a while, more and more classes and basic blocks are present on the mobile device and the callback rate declines.

If a callback to the server for what reason ever fails, SlimVM is not able to continue executing the program because parts of the bytecode are then missing. Therefore SlimVM needs to stop executing and shuts down. Not illustrated in Figure 4.10 is the data flow for requesting and receiving metainformation, but it follows the same schemata and is therefore negligible in that figure.

After all bytecode instructions for a given program are executed, completion ended successfully and execution of the program is done.

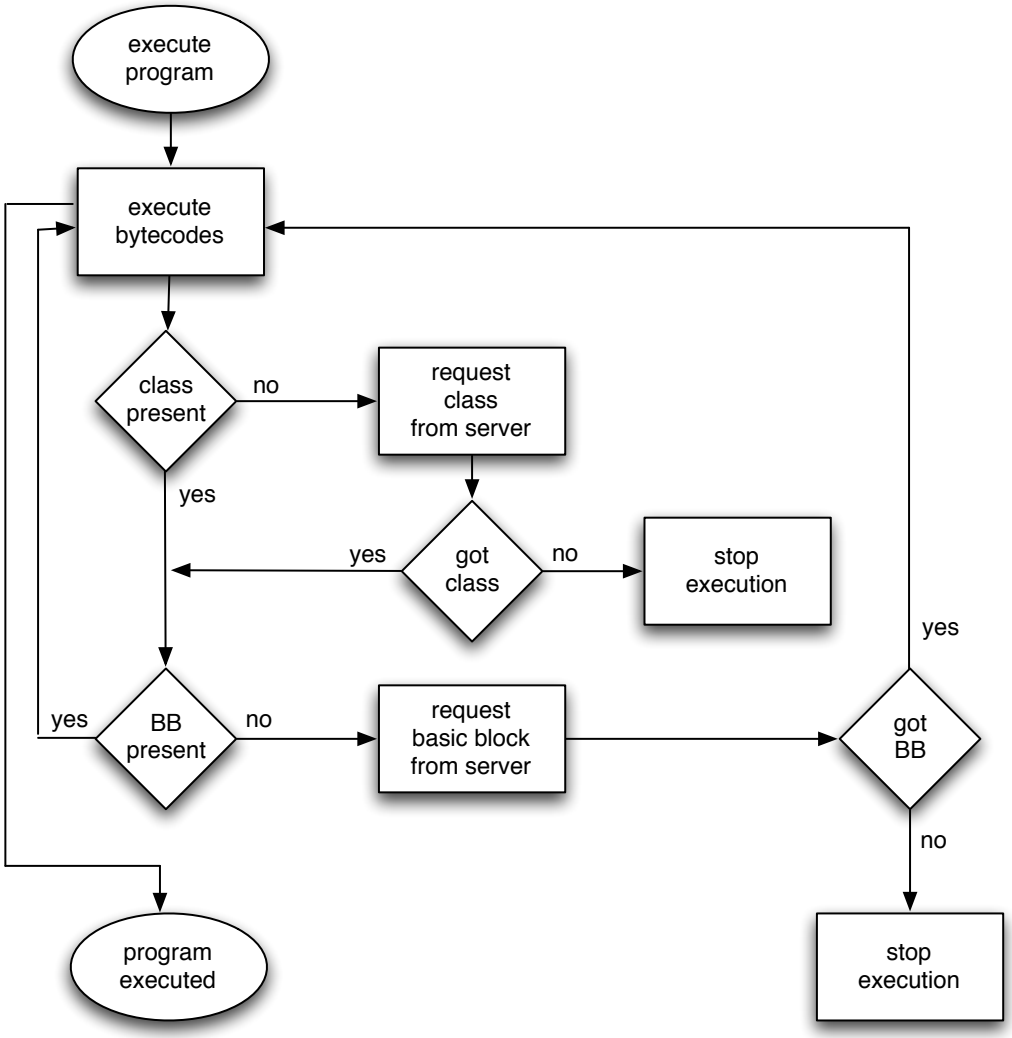

Figure 4.10: Data Flow in SlimVM.

Chapter 5

Implementation

This chapter gives in insight into the prototype implementation of SlimVM. Starting with the description of the development environment, this chapter presents detailed information about the Slimcompiler, which is used to compile Java class files into the new slim file format. Furthermore, this chapter contains detailed information about the callback functions to the server, modified opcodes, basic blocks as well as the Exceptionhandling.

5.1 Development Environment

The used software for developing this master's thesis consists of two mayor parts:

Eclipse
: Eclipse [IBM01] (version 3.3.2) is used as development environment. Eclipse is an open source framework, based on the Java technology, it is used for numerous types of software development. The most well known usage of Eclipse is as integrated development environment (IDE) for the programming language Java. By installing plugins written for the Eclipse software framework it can be extended for other programming languages.

  As the SlimVM prototype is programmed in C, for this master's thesis Eclipse was extended by such a plugin to offer the best available support to modify a Java virtual machine.

BCEL
: To extract information out of Java class files and to modify those class files, the Byte Code Engineering Library BCEL [Dah01] (version 5.2) was used. BCEL is a tool which provides a simple API for decompressing, modifying, and recomposing binary Java class files. BCEL was used to read and analyze Java class files and furthermore to compile the gathered information into the new slim file format.

5.2 Slim Compiler

The developed Slim compiler is based on the *Byte Code Engineering Library* BCEL and compiles Java class files into the new slim class file format. The basic idea is, to strip unnecessary information of a Java class file. Therefore, we replaced method, and class names, which are currently represented as string constants in the constant pool, by numeric identifiers. Hence, it is possible to drop the whole constant pool and to safe memory in a class file. In order to support the new class file format, a number of bytecode instructions need to be updated.

The slim compiler consists of two major data structures. The classmapper, which maps every class name to a numeric identifier and the methodmapper, which maps every method (including signature) to a numeric identifier.

First, classes like java.lang.String or java.lang.Class and methods like the method getSystemClassLoader, needed by SlimVM during startup, are loaded in the corresponding mapper, because these classes and methods need to have the same numeric identifier continuously, no matter what kind of application is loaded. Also primitive classes, like for example the primitive type int or the primitive type char need to have the same numeric identifier, because these classes have a fixed size and are needed to allocate memory in the Java virtual machine.

Table 5.1: Classmapper lookup table, which maps every class name to a numeric identifer

numeric identifier	**class name**
0	java.lang.Object
1	java.lang.Class
2	java.lang.String
3	java.lang.System
4	java.lang.Thread
5	java.lang.Throwable
6	java.lang.Error
...	...
59	(primitive type) byte
60	(primitive type) char
61	(primitive type) double
62	(primitive type) float
63	(primitive type) int
64	(primitive type) long
65	(primitive type) short
66	(primitive type) boolean
67	(primitive type) void
...	...
71	Factorial
...	...

Then, starting with the Java class file which includes the main() method, the slim compiler recursively analyzes all dependencies of these classes and maps every class and every method to a number. The slim compiler creates a lookup table for classes and methods, like the one mentioned in table 5.1. Further, the offsets for all static and object fields of a class are calculated and the information gathered is stored together for every class. This is necessary to replace the field resolution within the constant pool. Also the static and object size, which is needed by the Java virtual machine for allocating memory, is calculated during this phase. After that, the slim compiler starts analyzing the actual bytecode which is then segmented into basic blocks. This is done for every method. Basic blocks for every method are numbered consecutively and stored together with it. In the next step, all basic blocks are visited and some instructions are modified. Finally, the collected information (except of all bytecode instructions which remain on the network host) is written out in the new slim file format which is then stored on the server and waits to be requested for execution purpose by the mobile device.

5.3 Slim File Format

Since I analyze all Java class files on the server prior to execution on the client, I am able to change the representation of an original Java class file to my own, especially for this approach, customized slim file format. Unnecessary information is stripped and the representation of the constant pool is changed. Usually the constant pool is used to dynamically resolve symbolic references to fields, method and classes at run-time. But since I changed the representation of classes and methods from string constants to distinct numeric identifiers, the use of the constant pool is redundant for the new slim file format.

Different to a common Java class file, the new invented slim class file format has no section Access rights and no section Class Attributes. Usually the attributes section contains, amongst others, the bytecode instructions for every method of a class. But since the new format doesn't hold any actual bytecode instructions at all, we dropped these two sections. In other words, all bytecode instructions remain on the server in form of pre-linked basic blocks, the use for the attribute section became redundant. Figure 5.1 illustrates the new slim file format.

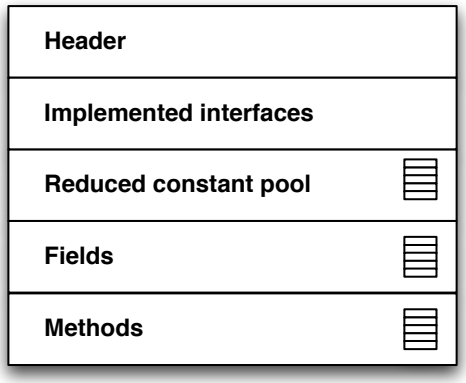

Figure 5.1: Slim file format.

Listing 5.1 presents the internal representation of the *Slim File Format* in detail. Equally to an original class file, also the slim class file represents exactly one class or interface and consists of a stream of 8-bit bytes where multibyte data items are stored in big-endian order. The * stands either for an internal used structure for *methods*, *fields*, the *reduced constantpool*, or *interfaces*.

Listing 5.1: Slim File Format

```
1  Class File {
2      u2 class_id:
3      u2 super_class_id;
4      u2 is_abstract;
5      u2 is_interface;
6      u1 has_main;
7      u2 main_id; (if has_main == 1;)
8      u1 has_clinit;
9      u2 clinit_id; (if has_clinit == 1;)
10     u2 interfaces_count;
11     * interfaces[interfaces_count];
12     u2 reduced_constant_pool_count;
13     * reduced_constant_pool[reduced_constant_pool_count];
14     u2 static_size;
15     u2 object_size;
16     u2 fields_count;
17     * fields[fields_count];
18     u2 methods_count;
19     * methods[methods_count];
20 }
```

The *Slim File Format* is divided into five basic sections which are described below:

Header

The header of the slim file format contains basic information of every class like the class_id and super_class_id. The header also contains information if the class is abstract or if the class is an interface, stored in is_abstract and is_interface. Further, the header includes information if the class has a main() method, and if so, the has_main value is followed by a number which represents the internal representation of this method stored in main_id. Finally, the header includes information if the class has a static constructor stored in has_clinit, and if so, this value is followed by a number which represents the internal representation of this method, stored in clinit_id.

Figure 5.2 illustrates the internal representation of the classid. Usually arrays are represented in the constant pool by a leading [followed by the name of the class. Since class names are no longer represented in form of string constants, I use the first 8 bit of the classid to indicate the array depth. The remaining 24 bit indicate the actual classid. This heads to a limitation for arrays with a depth deeper than 256, but since this doesn't happen often, this limitation is negligible for my approach.

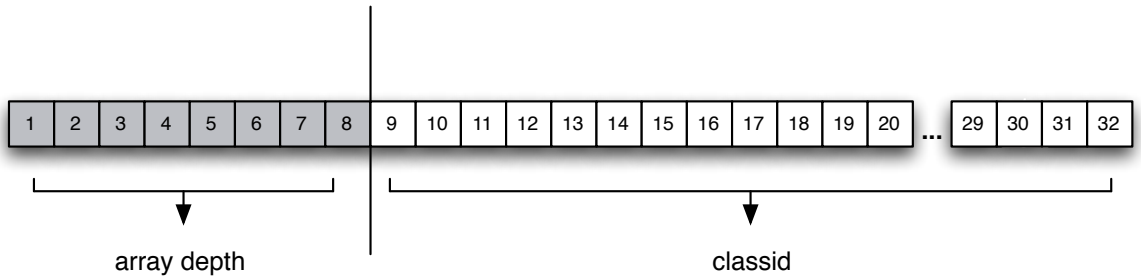

Figure 5.2: Internal representation of the classid.

Implemented interfaces

The interfaces_count indicates the number of direct implemented superinterfaces of the class followed by an array of classids, stored in interfaces[interfaces_count], which represent the implemented interfaces.

Reduced constantpool

Since I don't need the constant pool to dynamically resolve classes, methods and fields anymore, the constant pool almost became redundant. But only almost, because I have to transfer e.g. the Hello world! string constant, which should be displayed on the screen when executing a simple *Hello World* program. Therefore, I use the section Reduced constantpool to transfer string constants needed by a class file.

The reduced_constant_pool_count indicates the number of following string constants which are stored in the array reduced_constant_pool[reduced_constant_pool_count].

Fields

The section *Fields* holds information about the static and object size of a class, stored in static_size and object_size. Usually this values are calculated in the Java virtual machine during runtime, but as I changed the representation of the constant pool, I don't have the opportunity to look up any information in the constant pool. Therefore even constant values of fields are directly stored in the fields array.

The fields_count indicates the number of following fields which are stored in the array fields[fields_count].

Methods

The methods_count indicates the number of following methods of each class, which are stored in methods[methods_count]. For every method of a class, this section contains information like the distinct method identification number, whether the method is static or not and whether the method is native or not. If it is not , this section includes the number of basic blocks for the method. Otherwise the number of basic blocks is zero and the *Java virtual machine* calls back to the server to get the methods native name in order to execute this method.

Furthermore, this section includes information needed for execution in the Java virtual machine like max_stack, which indicates the maximum stack size or max_locals, which indicates the maximum number of local variables during the execution of this method. Finally, the section *methods* includes the number of parameters, which this method takes.

5.4 Server

As all information of the program to be executed is analyzed and stored in BCEL, I also used BCEL in order to develop the server for the SlimVM approach.

Immediately after the slim compiler has finished analyzing and writing out all referenced class files of the program to be executed, the server is started. All code, including class information in the new slim file format and bytecode instructions in form of prelinked basic blocks, remains on the server and is requested by the SlimVM on the client at runtime. In order to support this approach, the server needs to provide a number of callback functions which are presented in pseudo code in Listing 5.2.

5.4.1 Callback Functions

Listing 5.2: Pseudo code for the server

```
1  while(true){
2      switch(client_request){
3          getClassData();
4          getBasicBlock();
5          getMethodName();
6          getMethodId();
7          getClassName();
8          getClassId();
9          getFieldOffset();
10         getCatchBasicblockId();
11     }
12 }
```

The two most needed functions are the getClassData() and the getBasicBlock(). The first one is used to request class information in the new slim file format and the second one is used to request a piece of bytecode in form of a basic block. The other functions are needed for resolution purposes. Some of those functions are necessary to support reflection and native methods and others are necessary to support the exception handling in SlimVM. The following enumeration lists and describes the purpose of every callback-function provided by the server in more detail:

getClassData()
: Whenever the Java virtual machine on the client invokes a method, the *Linker* looks up the class of the method. If the class hasn't been transferred to the client so far, it is requested from the server. Therefore the client sends the numeric identifier of the class to the server, the server looks up the class and gets class in form of the new slim file format and sends the information requested back to the client in a bytestream.

getBasicBlock()
: This is possibly the function needed the most during execution of a program on the client. Whenever a method is invoked on the client, a callback to the server is necessary in order to request the first basic block of the method. This is done by sending the numeric identifier of class and method, and the number of the requested basic block. A basic block is only a subset of instructions of a method and therefore whenever a basic block reaches the end point of its instructions, either the following basic block, or, if it is a branch instruction, the corresponding basic block has to be requested from the server.

getMethodName()
: In order to support the *Java Native Interface* on the client, this function is needed to resolve a numeric identifier of a method and the identifier of the corresponding class, to a name of a method. The client sends these two numbers back to the server, which looks up the class in the classmapper and the method in methodmapper and sends the name of the method back to the client.

getMethodId()
This method is needed to support the JNI. The client sends the name of the method to the server in order to get the corresponding numeric identifier of the method back. The server does a lookup in the methodmapper and sends the correct number back to the client. With this number the Java virtual machine is able to invoke the correct method.

getClassName()
Similar to the method getMethodName(), this function helps the Java virtual machine on the client to resolve the numeric identifier of the class. The client sends the numeric identifier back to the server, which looks up the correct name in the classmapper and sends the corresponding name back to the client. This function is also needed to support the JNI.

getFieldOffset()
The purpose of this function is to support the Java Native Interface on the client. The client sends the identifier of the class, the field name and the field signature to the server in order to get the offset of this field back. The server looks up internally the requested field and sends the offset back to the client.

getCatchBasicblockId()
This function is needed to handle exceptions on the client. Since all bytecode instructions remain on the server and I dropped the whole Attribute section, also the *Exception table* is no longer part of the new class file format. The purpose of the exception table is to lookup the offset in order to handle the raised exception.

The Java virtual machine on the client calls this function giving the distinct class identifier, the distinct method identifier, the number of the basic block and the offset within that basic block where the exception was thrown. On the server, every modified instruction is mapped against the original instruction and therefore it is possible to send the number of the basic block which handles the raised exception back to the client. With the number back basic block which handles the exception, the client calls back to the server using the function getBasicBlock() in order to get the corresponding bytecode instructions for the exception handling and continuos execution.

5.5 Client

To support the new class file format and to load class information and basic blocks on demand, we changed the functionality of the class loader. SlimVM has no conventional class loader, it is rather more a network client which requests classes and basic blocks from the server and loads it into an internal used format. Certainly, the biggest changes affect the *interpreter*, because it has to interpret a lot of modified bytecode instructions.

5.5.1 Modified Instructions

In order to support the new slim class file format, we manipulated a lot of instructions. Since I dropped the whole constant pool in the new format, most of the manipulated instructions are instructions which reference into the runtime constantpool. The following enumeration lists and describes all modified instructions of the SlimVM approach:

Ldc, Ldc_w, Ldc2_w
Usually the ldc (load constant) opcode is followed by an index into the runtime constantpool, which resolves the constant to be loaded, before it is pushed onto the operand stack. As illustrated in Figure 5.3 the modified ldc opcode is followed by an identifier which is either:

- 1 for a string constant to be loaded onto the operand stack. If so, the next 4 bytes indicate the numerical class identifier and the last 4 bytes indicate the index into the reduced constant pool, where the string constant is stored.
- 2 for an int value. In this case, the next 4 bytes are redundant and the value to be pushed onto the operand stack is stored in the last 4 bytes.
- 3 for a float value. If so, the next 4 bytes are redundant and the value to be pushed onto the operand stack is stored in the last 4 bytes.

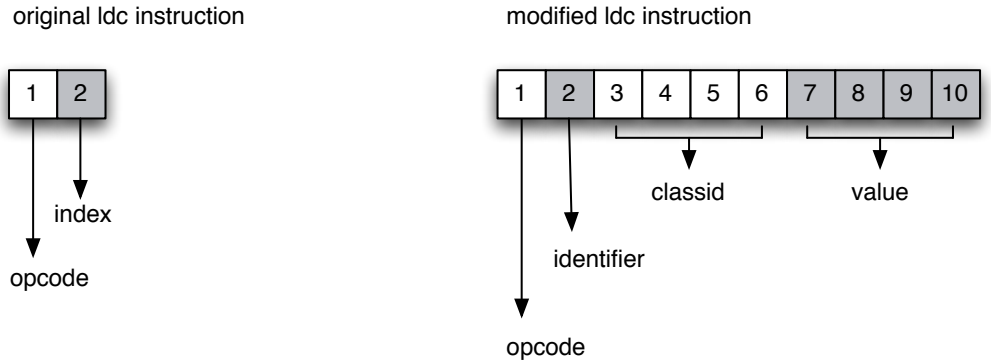

Figure 5.3: Comparison of original and modified ldc instruction used in SlimVM.

Getstatic
The modified getstatic opcode is followed by an identifier which indicates whether a value 8 bytes in size, like a double or float, or 4 bytes in size, like an int, is pushed onto the operand stack. The next 4 bytes indicate the distinct class identifier number, in which the field to be pushed is stored and the last 2 bytes indicate the field's offset.

Putstatic
Similar to getstatic, the putstatic opcode is followed by an identifier which indicates whether 8 or 4 bytes have to be popped from the operand stack and stored in the

field. The next 4 bytes indicate the numerical class identifier number of the field and the last 2 bytes indicate the fields offset within the class.

Getfield

The original getfield opcode is followed by 2 bytes, which indicate the index into the runtime constant pool in which the field is then resolved, and the value of it is pushed onto the operand stack. The modified getfield opcode, as illustrated in figure 5.4, is followed by an identifier which indicates whether the field is 8 bytes or 4 bytes in size and therefore whether 8 bytes (long, double) or 4 bytes (like an int) is pushed onto the operand stack. The last 2 bytes indicate the offset within the object.

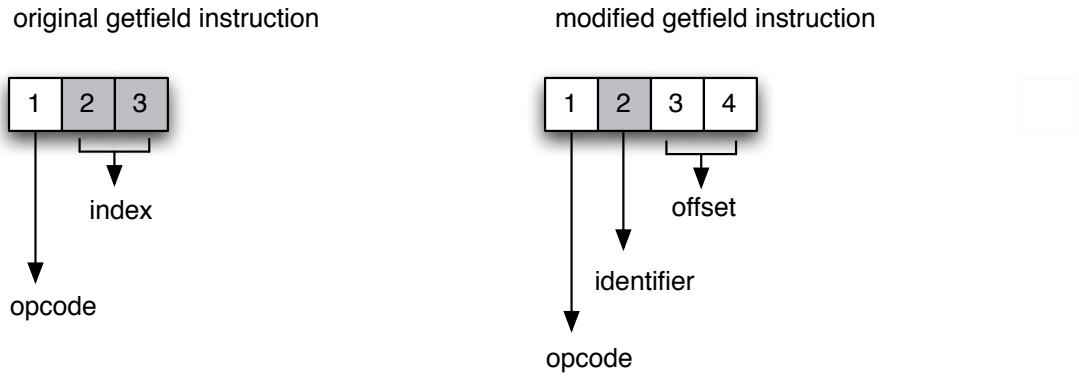

Figure 5.4: Comparison of original and modified getfield instruction used in SlimVM.

Putfield

Similar to the getfield instruction, the putfield opcode is followed by an identifier, which indicates whether 8 bytes or 4 bytes have to be popped from the operand stack and stored in the objects field. The following 2 bytes indicate the offset within the object.

Invokevirtual, **Invokespecial**, **Invokestatic**, **Invokeinterface**

The original opcode for invoking a method is followed by 2 bytes which indicate an index into the runtime constant pool in order to resolve the corresponding method. As illustrated in Figure 5.5 the modified instruction for a method invocation is followed by the numerical class identifier, the numerical method identifier and a basic block number. By means of this information, the method can be resolved. The modified class loader looks up the class and if the class has not been loaded so far, it is requested from the server. With the knowledge of the methods parameters, the virtual method lookup can be executed before the actual method is invoked and a new frame is popped on the frame stack. Finally, the correct basic block number is needed in order to request the correct block with the corresponding instructions so that the method can be executed.

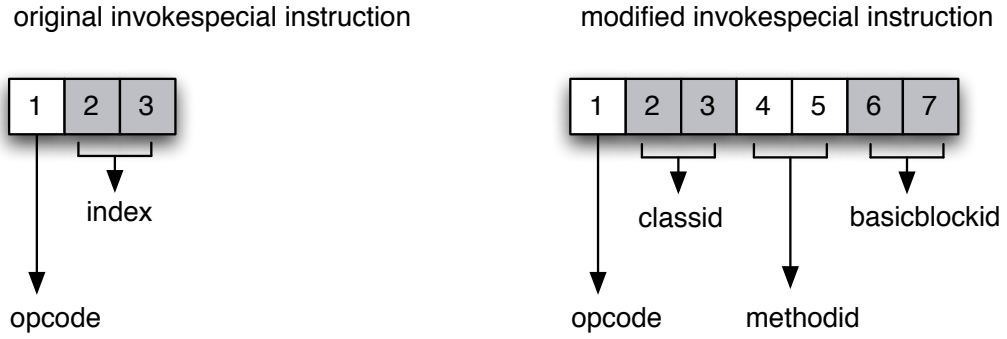

Figure 5.5: Comparison of original and modified invokespecial instruction used in SlimVM.

New

The opcode new is followed by the numerical class identifier which is used to resolve the correct class. When the class is resolved finally, an object of that class can be allocated and a reference to that class is popped onto the operand stack.

Anewarray

The opcode of the newarray instruction is almost identical to the new instruction. The opcode is followed by an numerical identifier which indicates the class of which an array should be allocated. A value is popped from the operand stack which indicates the array length. If that value is not negative, an array with the length of the popped value of the requested class is allocated and a reference is popped onto the operand stack.

Multianewarray

The multianewarray opcode is followed by a numerical class identifier and a number which indicates the dimensions of the array. The multi array is allocated and a reference to it is pushed onto the operand stack.

Branch Instructions

For all branch instructions like for example the goto instruction and all kinds of if instructions, the opcode is followed by an relative offset in the bytecode. As I segmented the bytecode into basic blocks, it is impossible to handle relative offsets within the bytecode. Therefore, as illustrated in figure 5.6, all offsets are replaced by a number of a basic block. With this information the correct basic block can be requested from the server and execution of the program continues.

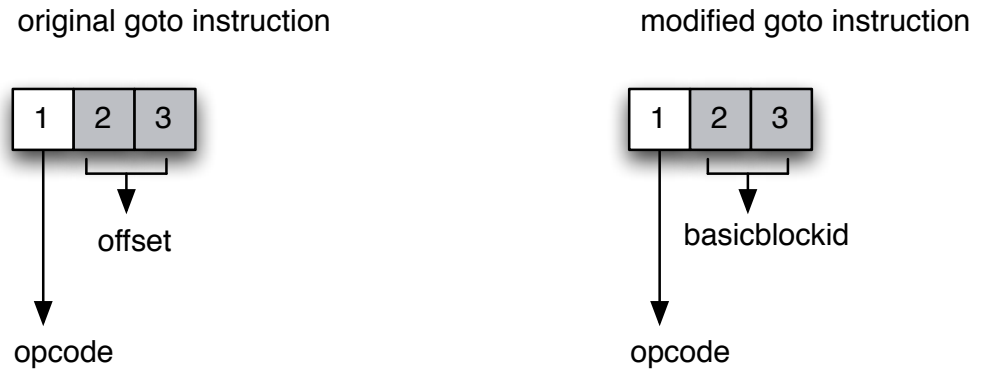

Figure 5.6: Comparison of original and modified goto instruction used in SlimVM.

5.5.2 Basic blocks

A basic block is a sequence of instructions with only one entry and one exit point. That means, no jump instructions and also no destinations of jump instructions are included within a basic block.

Whenever a basic block is transferred to the mobile device, the client adds a goto instruction at the end. This is reasonable, because whenever a basic block reaches its end, the next basic block has to be requested from the server. To overcome the overhead, this is done on the client, because it is useless to transfer an additional instruction for every basic block when this can be done on the client.

Figure 5.7 illustrates the breakdown of bytecode to basic blocks of the factorial program given in Listing 2.6. This example also uses the modified instructions for the SlimVM approach.

The first two instructions iconst_1 and istore_0 are the same as in the original bytecode. The first modified instruction is getstatic at address 2. Classid 73 is the numerical identifier of the class Factorial(), identifier 2 indicates that the field holds an integer value at offset 0. Basic block 1 doesn't include any changes compared to the original bytecode.

Basic block 2 at address 21 shows an ifgt instruction which is now followed by the basic block number 1. Is the value of the local variable in slot 1 greater than 0, than a jump to basic block 1 is executed.

Basic block 3 starts with a getstatic instruction. Classid 3 is the numerical identifier for the class java.lang.System, identifier 5 indicates that the field at offset 4 is a reference to an other class. The next interesting instruction in this basic block is found at address 32. Invokevirtual of classid 74 and methodid 34 means that the method java.io.PrintStream.println should be invoked. The branch instruction at address 39 has the number 5 as an operand which indicates a jump to basic block 5 and the method returns.

Basic block 4 includes all instructions necessary for the exception handling. Is no exeception thrown, this code is never executed. At address 43, the field java.lang.System.err is popped onto the operand stack and at address 51 the method java.io.PrintStream.println

is invoked, before the method in basic block 5 finally returns.

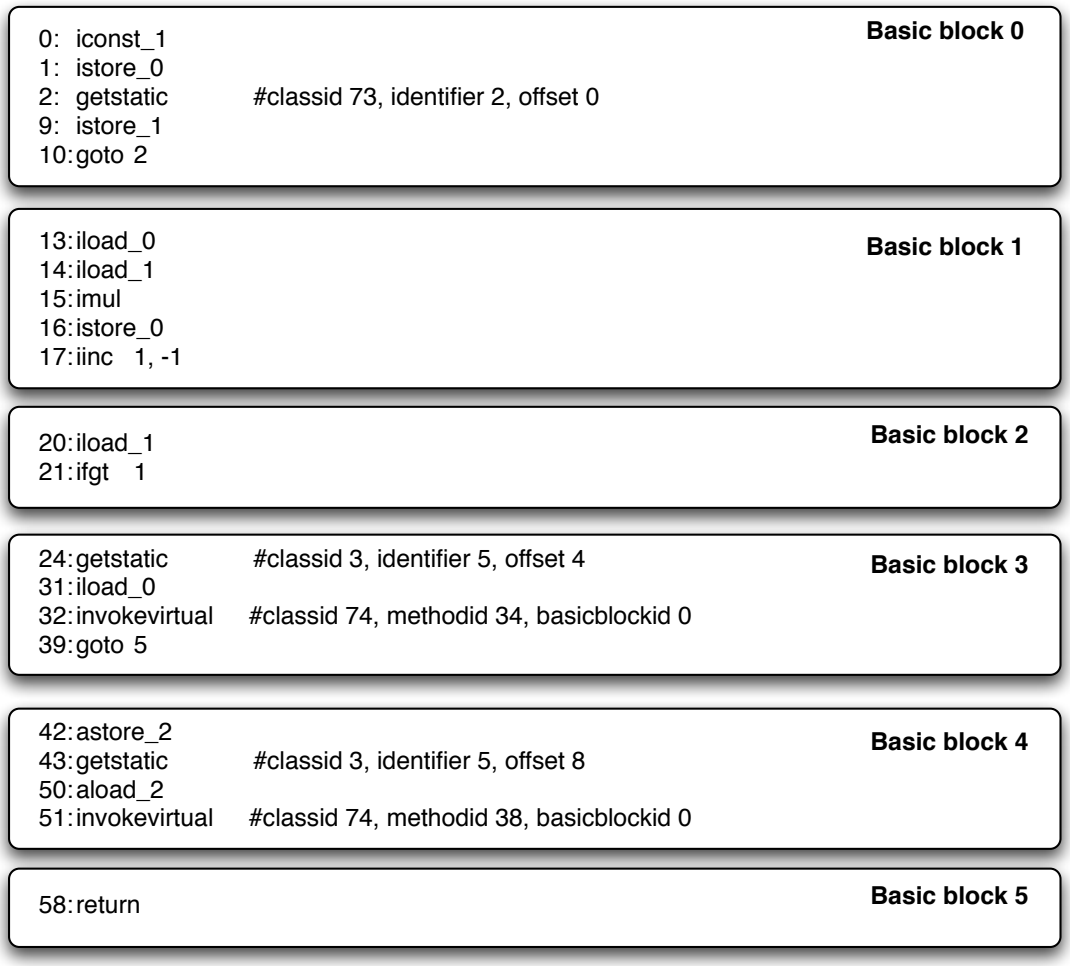

Figure 5.7: Basicblock example.

5.5.3 Exceptiohandling

To support *Excpetionhandling* in the SlimVM approach, I mapped every modified instruction against its original instruction. This mapping table (Table 5.2) is stored on the server. Whenever an exception occurs in the Java virtual machine on the client, it calls back to the server using the function getCatchBasicblockId(). Given the numerical identifier of the class, the identifier of the method and the number of the basic block including the offset where the exception occured, the server looks up the instruction and gets the address of the original instruction. With this address a lookup in the original *Exception table* is performed, which gives the target address back. With this address, a lookup is performed

in the mapping table which gives the number of the basic block back.

Table 5.2: Mapping table to support *Exceptionhandling* in SlimVM

original instructions	**modified instructions**	
0: iconst_1	0: iconst_1	Basic block 0
1: istore_0	1: istore_0	
2: getstatic	2: getstatic	
5: istore_1	9: istore_1	
6: goto 16	10: goto 2	
9: iload_0	13: iload_0	Basic block 1
10: iload_1	14: iload_1	
11: imul	15: imul	
12: istore_0	16: istore_0	
13: iinc 1, -1	17: iinc 1, -1	
16: iload_1	20: iload_1	Basic block 2
17: ifgt 9	21: ifgt 1	
20: getstatic	24: getstatic	Basic block 3
23: iload_0	31: iload_0	
24: invokevirtual	32: invokevirtual	
27: goto 38	39: goto 5	
30: astore_2	42: astore_2	Basic block 4
31: getstatic	43: getstatic	
34: aload_2	50: aload_2	
35: invokevirtual	51: invokevirtual	
38: return	58: return	Basic block 5

Exception table:
from to target type
20 27 30 Class java/lang/Exception

For example, when the instruction invokevirtual of basic block 3 raises an exception, the client calls back to the server with the corresponding class and method identifier. A lookup in this table is performed where the modified instruction invokevirtual of address 32 is mapped against the original instruction at address 24. With address 24, a lookup in the *Exception table* is performed and the the target for the handling at address 30 is found. Again, a lookup in the mapping table is performed. Address 30 is the entry point of basic block 4 and therefore the number 4 is sent back to the client. With this number, the client requests the bytecode instructions of this basic block and execution of the Java program continues.

Chapter 6

Evaluation

In this chapter, the results of the gathered tests are given. The goal of the following performance tests is to show that only a small subset of information of an Java class file is actually needed for execution of a program, even though conventional Java virtual machines load the whole class.

6.1 Test Environment

The test environment for the prototype implementation of SlimVM consists of two separate running systems, the client and the server. Client and server are connected over the router Zyxel - Prestige 334 with a 100MBit ethernet cable.

Server
On the server side we use the *Byte Code Engineering Library* BCEL version 5.2 for analyzing Java class files. The actual server is also implemented using BCEL and runs on an Intel Pentium 4 PC with 2.53 GHz CPU and 512 MB RAM. The used operating system for the server is Microsoft Windows XP Professional Version 2002.

Client
The client consists of an Apple MacBook with 2 GB 667 MHz DDR2 SDRAM and a 2 GHz Intel Core 2 Duo. The used operating system on the MacBook is Mac OS X Leopard version 10.5.4. For the developed SlimVM, we modified and customized the Java virtual machine JamVM version 1.5.1.

6.2 Overview of the Benchmarks

We use a number of test programs and some selected benchmarks in order to be able to compare the gathered results by the developed SlimVM. We run all tests on SlimVM as well as on JamVM, hence, it is possible to compare the gathered results. Furthermore, these test cases are used to present the achieved space savings and the effect on the performance of the prototype implementation.

We use a program with an empty main method, which is meant to pinpoint the startup overhead of SlimVM. Furthermore, we use a simple HelloWorld program and the afore mentioned and discussed Factorial program.

In order to be able to make some standardized and comparable statements for programs executed by SlimVM, we use the following benchmarks:

Linpack [Jec04]
performs numerical linear algebra like vector and matrix operations.

Scimark 2 [PM04]
is a benchmark which measures numerous computational kernels and summarizes the score in approximate Mflops/s.

FloatingPointCheck of **Specjvm98** [SPE98]
measures a number of floating point operations and presents a summarized score of the gathered results.

ArithBench of **JavaGrande** [EPC07]
is a benchmark which measures a number of arithmetical operations

LoopBench of **JavaGrande** [EPC07]
is a benchmark which measures a number of loop operations

6.3 Measured Results

One of the initial claims of this thesis is, to demonstrate the load overhead of a Java virtual machine. In the next section we present the space savings achieved by off target analyzing of Java class files, pre linking and on demand code loading.

The measurements show the ratio between the information analyzed on the server compared to the information transferred to the client. We do not compare the information transferred to the whole gnu classpath which consists of 7.258 class files with a total size of 14.89 MB (15.613.297 bytes).

6.3.1 Memory Space Savings

Figure 6.1 shows the ratio between the analyzed information in bytes and bytes transferred to the client. Conventional Java virtual machines need to have an enormous amount of library code to be present on each client device. SlimVM, on the other side, loads only parts of a class file, necessarily needed for execution.

As illustrated in Figure 6.1, commonly 94% of information analyzed on the server is not needed for execution of a Java program. That means, even if only needed class files are present on a conventional mobile device, that 94% of the information stored in those class files is not needed for the correct execution of that Java program.

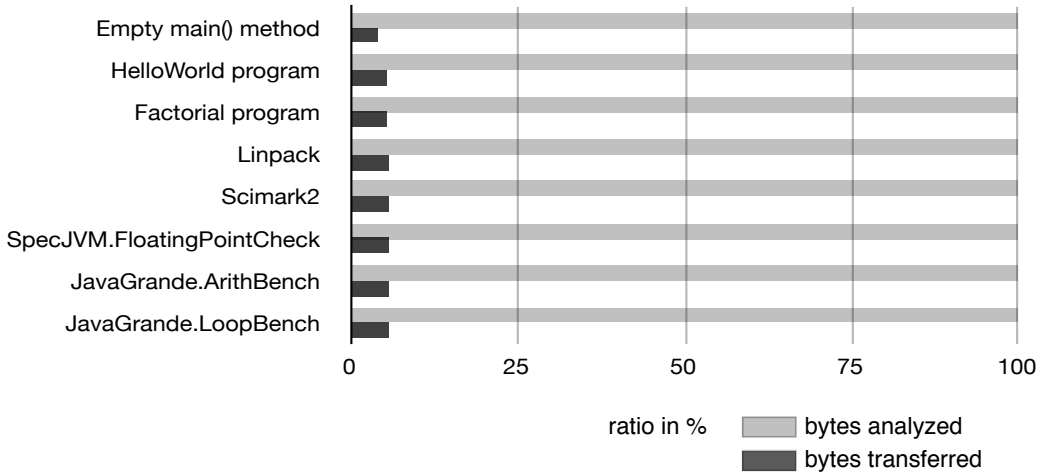

Figure 6.1: *The ratio between bytes analyzed on the server and bytes transferred to the client shows, that commonly only 5% of the analyzed bytes on the server are actually requested by SlimVM on the client.*

The exact numbers of bytes analyzed on the server and bytes transferred to the client are listed in more detail in Table 6.1.

Table 6.1: Analyzed and transferred bytes of performed test cases

	bytes analyzed	**bytes transf.**	**Reduction**
Empty main() method	2.979.103	119.376	95.9%
HelloWorld program	2.979.264	163.783	94.5%
Factorial program	2.979.595	163.857	94.5%
Linpack	2.984.922	168.339	94.3%
Scimark2	3.000.434	172.269	94.2%
SpecJVM.FloatingPointCheck	2.992.489	172.191	94.2%
JavaGrande.ArithBench	2.993.179	174.782	94.1%
JavaGrande.LoopBench	2.988.503	170.009	94.3%

One of the most interesting facts is, that a common JVM loads a high amount of a programs bytecode into memory, but in fact, only few bytecode instructions are needed to execute that program.

Measurements conducted show, that usually 97% of bytecode instructions in form of basic blocks are not needed for execution of that program. Figure 6.2 illustrates this fact which is listed in detail in Table 6.2.

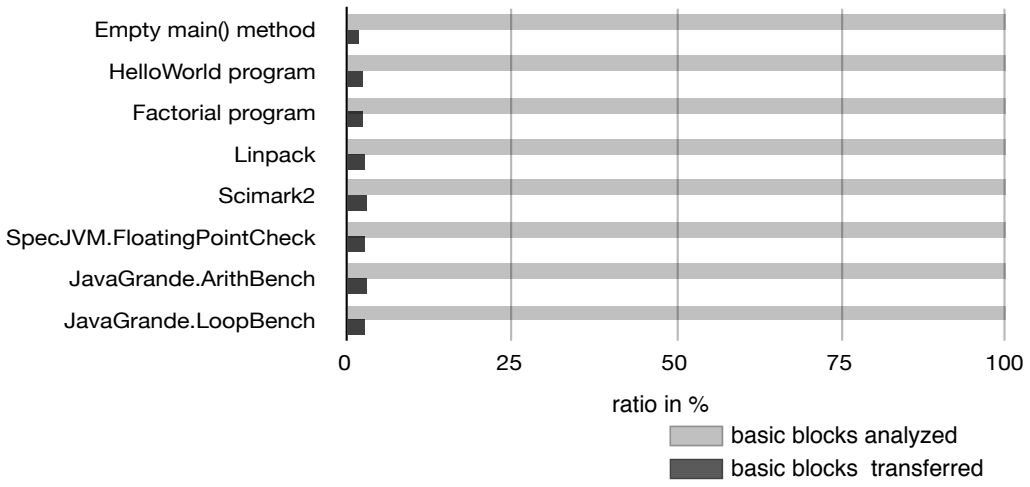

Figure 6.2: *The ratio between analyzed basic blocks on the server and basic blocks transferred to the Slim virtual machine on the client show, that a common Java program needs, on average, only 3% of the analyzed basic blocks for the correct execution of a Java program.*

Table 6.2: Ratio between analyzed and transferred basic blocks

	BB analyzed	**BB transferred**	**Reduction**
Empty main() method	38.842	796	98.0%
HelloWorld program	38.842	1.044	97.3%
Factorial program	38.849	1.048	97.3%
Linpack	38.991	1.166	97.0%
Scimark2	39.189	1.225	96.8%
SpecJVM.FloatingPointCheck	38.950	1.122	97.1%
JavaGrande.ArithBench	39.037	1.218	96.8%
JavaGrande.LoopBench	38.951	1.131	97.0%

Figure 6.3 shows the ratio between classes analyzed on the server and classes requested by the Java virtual machine on the client. As listed in Table 6.3, usually 84% of the classes analyzed on the server are not requested by SlimVM on the client. Compared to all class files of the GNU classpath which consists of 7.258 classes, only 195 classes are actually needed to execute a simple HelloWorld program.

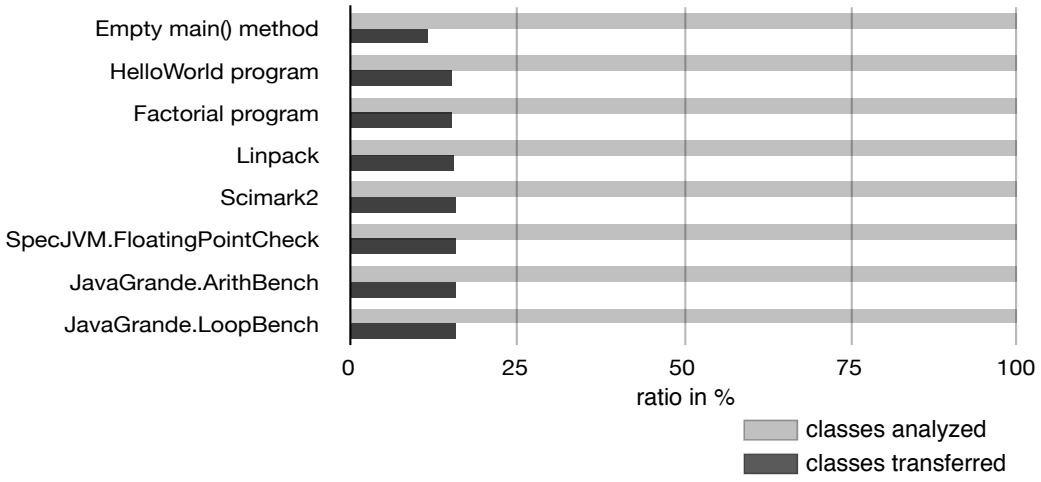

Figure 6.3: *Ratio between classes analyzed on the server and classes transferred to the mobile device show, that commonly only 15% of the classes analyzed are actually transferred to the client.*

Table 6.3: Ratio between analyzed and transferred classes

	classes analyzed	**classes transferred**	**Reduction**
Empty main() method	1.269	150	88.2%
HelloWorld program	1.269	195	84.6%
Factorial program	1.269	195	84.6%
Linpack	1.269	198	84.3%
Scimark2	1.277	206	83.8%
SpecJVM.FloatingPointCheck	1.274	205	83.9%
JavaGrande.ArithBench	1.273	204	84.0%
JavaGrande.LoopBench	1.273	203	84.1%

6.3.2 Memory Space Savings for Lazy Loading of Classes

Figure 6.4 illustrates the ratio between the size of classes loaded by JamVM compared to the size of requested slim classes (including the size of requested basic blocks during runtime) loaded by SlimVM. Measurements show that commonly 70% of the size loaded by a common Java virtual machine is not necessarily needed for the correct execution of that program.

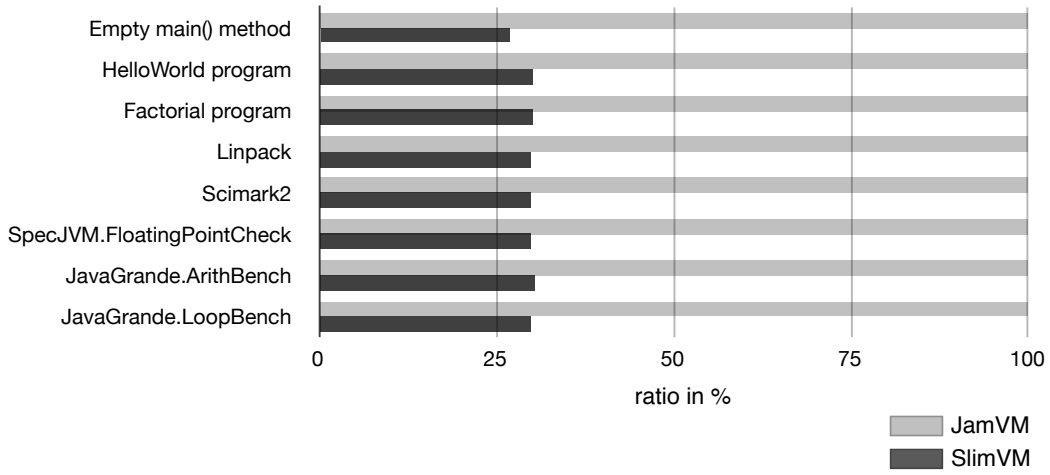

Figure 6.4: *The ratio between the size of classes loaded by JamVM and the size of classes (including requested basic blocks) loaded by SlimVM shows, that commonly 70% of the bytes loaded by a common JVM is not necessarily needed for execution of that program.*

Table 6.4: Ratio between bytes loaded by JamVM and bytes loaded by SlimVM

	JamVM	**SlimVM**	**Reduction**
Empty main() method	443.837	119.376	73,1%
HelloWorld program	543.414	163.783	69,8%
Factorial program	543.745	163.857	69,8%
Linpack	560.869	168.339	69,9%
Scimark2	575.533	172.269	70,0%
SpecJVM.FloatingPointCheck	577.587	172.191	70,1%
JavaGrande.ArithBench	574.318	174.782	69,5%
JavaGrande.LoopBench	569.642	170.009	70,1%

6.3.3 Class File Size Reduction

Since we use our own slim file format for transferring classes, we are able to reduce the size of data transfers to the client. As mentioned in the previous chapters, we drop unnecessary information and use our own format for field, method, and class resolution. Hence, as illustrated in Figure 6.5, we are able to reduce the file size of a Java class file by about 85%, but with the annotation that these class files do not include any actual bytecode instructions, as those are stored seperately on the server. Detailed information of space savings for all executed test programs is given in Table 6.5.

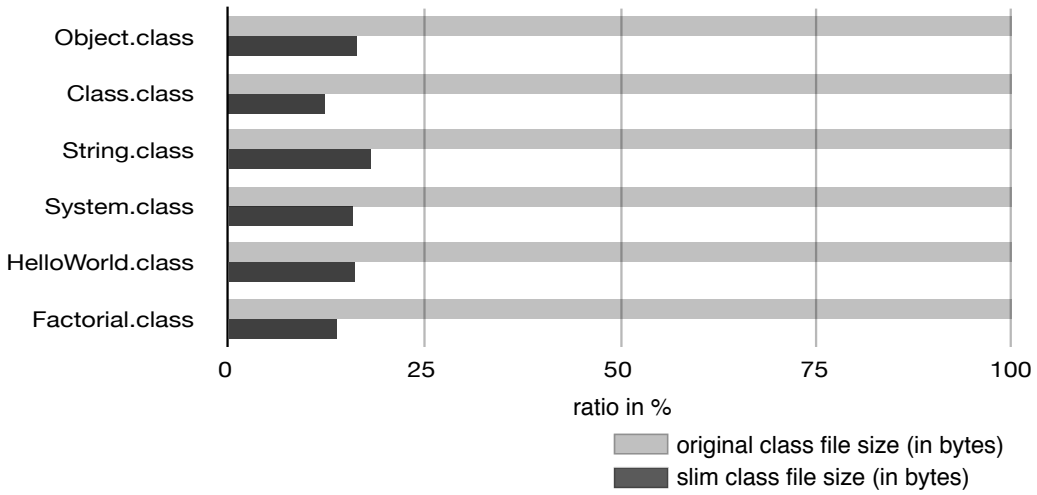

Figure 6.5: *The ratio between the size of original class files compared to the slim files shows, that a slim file only has 15% of the size of an original class file, but with the annotation that the slim files do not include any actual bytecode instructions.*

Table 6.5: Size reduction of class files in the new slim file format

	original class size	**slimvm class size**	**Reduction**
Object.class	1.931	320	83.4%
Class.class	15.719	1.949	87.6%
String.class	15.645	2.873	81.6%
System.class	4.160	672	83.8%
HelloWorld.class	542	89	83.5%
Factorial.class	873	123	85.9%

6.3.4 Effect on Performance

The initial claim of SlimVM was to reduce the code footprint for connected embedded Java virtual machines. We present a prototype implementation with a simple execution mode where no optimization work for increasing the execution speed was done. The bottleneck is the network delay which slows down the execution speed of SlimVM. Furthermore, we did not implement opcode rewriting or any other optimization which would make the presented virtual machine competitive with a conventional JVM.

Table 6.6 and table 6.7 list the values of the ArithBench and Loopbench of the JavaGrande benchmark. The resulsts of the ArithBench for addition, multiplication and division of values are almost identic. The results of LoopBench show, that SlimVM is slightly slower than JamVM for loops. This is most possibly caused by the basic block jumps. An conventional Java virtual machine is able to make jumps for and back in the bytecode, SlimVM has to a look up first, if the corresponding basic block for a jump instruction has been loaded so far. If so, the basic block is loaded and the instructions of it are executed, otherwise the basic block has to be requested from the server. These two activities slow down the SlimVM for loop operations.

Table 6.6: ArithBench of JavaGrande

	JamVM	**SlimVM**	
Section1:Arith:Add:Int	5.0050404E7	4.2522708E7	(adds/s)
Section1:Arith:Add:Long	3.3375432E7	3.7372264E7	(adds/s)
Section1:Arith:Add:Float	4.8466204E7	4.5153644E7	(adds/s)
Section1:Arith:Add:Double	3.3506826E7	3.554784E7	(adds/s)
Section1:Arith:Mult:Int	5.1220008E7	3.9982916E7	(multiplies/s)
Section1:Arith:Mult:Long	3.0621438E7	3.3924836E7	(multiplies/s)
Section1:Arith:Mult:Float	4.7202536E7	4.4667392E7	(multiplies/s)
Section1:Arith:Mult:Double	3.4351608E7	3.3895008E7	(multiplies/s)
Section1:Arith:Div:Int	3.9701944E7	2.9483534E7	(divides/s)
Section1:Arith:Div:Long	1.766469E7	2.0736616E7	(divides/s)
Section1:Arith:Div:Float	4.2412632E7	4.4158748E7	(divides/s)
Section1:Arith:Div:Double	3.477818E	3.4905992E77	(divides/s)

Table 6.7: LoopBench of JavaGrande

	JamVM	**SlimVM**	
Section1:Loop:For	4.8624424E7	1.8106868E7	(iterations/s)
Section1:Loop:ReverseFor	6.5392136E7	2.0531328E7	(iterations/s)
Section1:Loop:While	5.266474E7	2.7598754E7	(iterations/s)

6.3.5 Runtime Distribution of Code requested

Figure 6.6 and Figure 6.7 show the runtime distribution of code requested from the server. As illustrated, the code requested for a simple Hello world program is almost continuously where else the code requested for the FloatingPointCheck is high in the beginning, but then almost identic. This is due to the while loop which is performed in this benchmark.

The big jump of requested code in both programs is the request of the class gnu.java.lang.CharData which is still enormous in size.

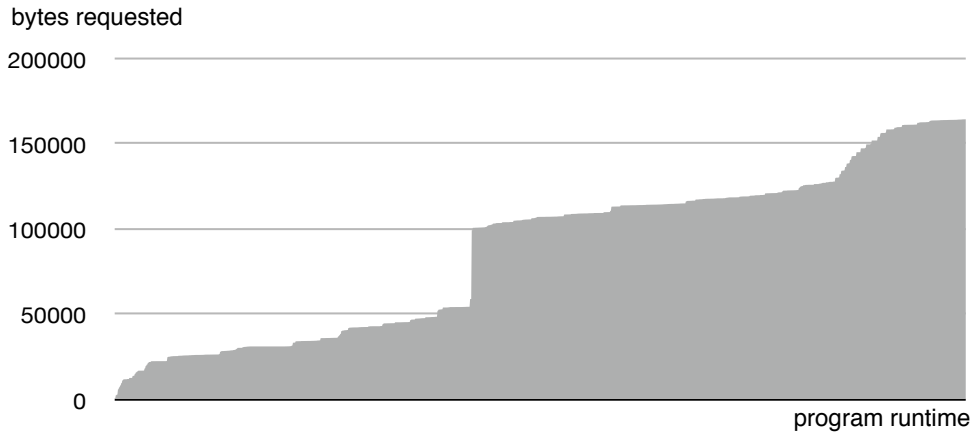

Figure 6.6: *Runtime distribution of bytes requested from the server for a HelloWorld program.*

Figure 6.7: *Runtime distribution of bytes requested from the server for the FloatingPointCheck benchmark for SpecJVM.*

6.3.6 Appearance of ldc Instructions

Table 6.8 gives an overview of the number of appearances of a ldc instruction for int, float, long or double. This table shows that the appearance of such an instruction is very low and even when it appears is the likelihood that one and the same constant value out of the constant pool can be used for two or more instructions is very low. That means, that the overhead for transferring a constant value directly with the bytecode instruction is negligible.

Table 6.8: Appearance of ldc in the most common classes

	int	**float**	**long double**	**total**	**identic**
Object.class	1	0	0	0	0
Class.class	3	0	0	0	0
String.class	4	0	0	0	0
System.class	0	0	0	0	0
Thread.class	3	0	0	3	2

Chapter 7

Conclusions

In the preceding chapters, the design decisions and the various techniques of the SlimVM were presented. Finally, this chapter concludes the work of this master's thesis by summarizing the main contributions including their limitations. Last but not least, some future work to improve and optimize the approach of SlimVM is discussed.

In this master's thesis we presented a solution for persistent connected embedded Java virtual machines where all code, including application and library code, resides on a network host and is requested by the client only on demand.

This master's thesis revolutionalizes present research projects in two important ways. On the one hand, only parts of a Java class file are needed for the correct execution of that program, and on the other hand, something else than *String constants* can be used for field, method and class resolution within a JVM.

Since all Java class files needed for execution of that Java program are analyzed prior the execution on the client, we are able to use our own slim file format for Java classes. Dropping unnecessary information and customizing the way of field, method and class resolution makes it possible to reduce the class file size.

The breakdown of Java bytecode into basic blocks makes it possible, that only instructions actually executed, are loaded by the Java virtual machine on the client. Hence, no overhead of bytecode instructions has to be loaded by the VM.

We developed a prototype implementation of SlimVM with a simple execution mode and therefore here are some ideas, which would increase the execution speed of SlimVM and would make the approach of SlimVM more competitive. At the same time, this ideas are part of the future work for optimizing the approach of SlimVM.

A global data structure for basic blocks, not separately for every method as in the current prototype implementation could increase the lookup speed on client and server and reduce the network transfer. Furthermore, some of the reserved opcodes could be used to innovate some new opcodes. For example, the ldc opcode currently is followed by an identifier which is used to identify whether the following constant is a string, integer or long value. In this case, three different opcodes could be used in order to increase the execution speed. Furthermore, it would be interesting how often the same string constants

are transferred in different classes. The usage of a global constant pool could reduce the data transfer to the mobile device. Heuristic has to be done in order to discover how often one and the same string constant is transferred in the current prototype. Finally, heuristics could help in order to send more basic blocks at the same time. For example, a certain basic block is requested, and whenever this basic block is requested, also another certain basic block is requested right after that basic block. Hence, these two basic blocks can be transferred at the same time and therefore the JVM on the client does not have to call back to the server again, to request that basic block.

Summing up, we presented a Java virtual machine for connected embeddes systems, which show that only a small subset of information is actually needed for the correct execution of a Java program.

Appendix A

Definitions

AC	Agglomerative Clustering
AG	Abstract Grammar
API	Application Programming Interface
AST	Abstract Syntax Tree
BB	Basic Block
BCEL	Byte Code Engineering Library
CAP	Converted Applet
CPU	Central Processing Unit
DLL	Dynamic Link Library
GC	Garbage Collector
GNU	Gnu's Not Unix
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
JAR	Java Archive
JIT	Just in time
JNI	Java Native Interface
JVM	Java Virtual Machine
KB	Kilobyte
LIFO	Last in first out
MB	Megabyte
MIDP	Mobile Information Device Profile
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SDRAM	Synchronous Dynamic Random Access Memory
TCP	Transmission Control Protocol
VM	Virtual Machine

Bibliography

[AP98] Denis N. Antonioli and Markus Pilz. Analysis of the Java Class File Format. Technical report, Department of Computer Science, University of Zurich, 1998.

[BHV98] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An Efficient Compressed Format for Java Archive Files. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1998.

[BS98] Boris Bokowski and André Spiegel. Barat - A Front-End for Java. Technical Report B-98-09, Institut für Informatik, Freie Universität Berlin, December 1998.

[Cla97] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. In *Practice and Experience Vol. 9 (11)*, pages 1031–1045, 1997.

[CSCM00] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java Bytecode Compression for Low-End Embedded Systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, 2000.

[Dah01] Markus Dahm. Byte Code Engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, April 2001.

[Deu96] Peter Deutsch. GZIP file format specification version 4.3. `http://www.ietf.org/rfc/rfc1952.txt`, May 1996.

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code compression. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 358–365, New York, NY, USA, 1997. ACM.

[EPC07] EPCC. The Java Grande Forum Benchmark Suite. `http://www.epcc.ed.ac.uk/research/activities/java-grande/`, 2007.

[FK97] Michael Franz and Thomas Kistler. Slim Binaries. *Commun. ACM*, 40(12):87–94, 1997.

[GNU99] GNU. GNU Classpath. `http://www.gnu.org/software/classpath/`, 1999.

[Hag01] Peter Hagger. Understanding bytecode makes you a better programmer. `http://www.ibm.com/developerworks/ibm/library/it\-haggar\_bytecode`, 2001.

[IBM01] IBM. Eclipse. `http://www.eclipse.org/`, 2001.

[Jec04] Mario Jeckle. Linpack Benchmark – Java Version. `http://www.jeckle.de/freeStuff/jLinpack/index.html`, 2004.

[LF03] Mario Latendresse and Marc Feeley. Generation of fast interpreters for Huffman compressed bytecode. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 32–40, New York, NY, USA, 2003. ACM.

[Lia99] Sheng Liang. *The Java Native Interface - Programer's Guide and Specification*. The Java Series. Addison-Wesley, 1999.

[Lou04] Robert Lougher. JAMVM. `http://jamvm.sourceforge.net/`, July 2004.

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

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. The Java Series. O'Reilly, 1997.

[NG96] Mark Nelson and Jean-Loup Gailly. *The Data compression book*. M&T Books, 1996.

[PKW89] PKWARE. .ZIP File Format Specification. `http://pkware.com/documents/casestudies/APPNOTE.TXT`, 1989.

[PM04] Roldan Pozo and Bruce Miller. SciMark 2.0 Benchmark. `http://math.nist.gov/scimark2/`, 2004.

[Pug99] William Pugh. Compressing Java Class Files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 247–258, New York, NY, USA, 1999. ACM.

[RMH99] Derek Rayside, Evan Mamas, and Erik Hons. Compact Java Binaries for Embedded Systems. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 9. IBM Press, 1999.

[SHF00] Christian H. Stork, Vivek Haldar, and Michael Franz. Generic Adaptive Syntax-Directed Compression for Mobile Code. Technical report, Department of Information and Computer Science, University of California, Irvine, November 2000.

[SMBZ07] Dimitris Saougkos, George Manis, Konstantinos Blekas, and Apostolos V. Zarras. Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments. *IEEE Transactions on Software Engineering*, 33, July 2007.

[SPE98] Standard Performance Evaluation Corporation SPECJVM. SPEC JVM98 Benchmarks. `http://www.spec.org/jvm98/`, 1998.

[ST98] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 324–332, New York, NY, USA, 1998. ACM.

[Sun99a] Sun. JAR File Specification. `http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html`, 1999.

[Sun99b] Sun. The K virtual machine (KVM) white paper. `http://java.sun.com/products/cldc/wp/`, 1999.

[Sun08a] Sun. Java Card Technology. `http://java.sun.com/javacard/`, 2008.

[Sun08b] Sun. Mobile Information Device Profile (MIDP). `http://java.sun.com/products/midp/`, 2008.

[Sun08c] Sun. Pack200 Compression. `http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/pack200.html`, 2008.

[TSL+02] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical Extraction Techniques for Java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.

[Wag07] Gregor Wagner. SlimVM: Optimistic Partial Program Loading for Connected Embedded Java Virtual Machines. Master's thesis, Technical University Graz, April 2007.

[WGF08] Gregor Wagner, Andreas Gal, and Michael Franz. Slim VM: optimistic partial program loading for connected embedded Java virtual machines. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 117–126, New York, NY, USA, 2008. ACM.

[Wik08] Wikipedia. Java class library. `http://en.wikipedia.org/wiki/Java\_Class\_Library`, November 2008.

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.