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

Christoph Kerschbaumer*† Gregor Wagner* Christian Wimmer*
Andreas Gal* Christian Steger† Michael Franz*

*University of California, Irvine †Technical University Graz, Austria
{ckerschb, wagnerg, cwimmer, gal, franz}@uci.edu steger@tugraz.at

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 and memory footprint to ensure Java's success on such systems. SlimVM's approach eliminates all unnecessary code and meta information on mobile devices.

We present a solution for the next generation of mobile computing environments for persistent connected embedded systems where all code resides on a network server and is requested at run time by the Java virtual machine on the client. 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. Java bytecode is manipulated and transferred to the client in the form of pre-linked basic blocks. Measurements show a reduction of the memory footprint of up to 70%.

Categories and Subject Descriptors

D.3.4. [**Programming Languages**]: Processors-Incremental compilers; optimization; runtime environments

General Terms

Design, Optimization, Performance, Experimentation

Keywords

Java virtual machine, optimization, connected embedded systems, code-size reduction, just-in-time compilation

1. INTRODUCTION

The efficient use of a Java virtual machine as an execution environment has become popular over the last decade. The idea that a program does not have to be compiled into machine code for every single platform but can be executed by a virtual machine on the target device has contributed to the success of Java virtual machines. The use of virtual machines is not restricted to personal computers and laptops. They are also very popular on mobile devices like cellular phones and PDAs.

However, the memory footprint of a full Java system is often too big for such devices. Reduced JVMs with smaller specifications like the *Java Platform, Micro Edition* [23] contain only a small subset of the class library, and are therefore not able to execute standard Java applications.

Instead of limiting the Java specification, we present a client-server approach for mobile devices where all Java code resides on a network server. Only code that is actually executed is transfered to the client on demand and then executed by a full-fledged Java VM. A significant part of a Java class is consumed by symbolic names required for the linking process when the class is loaded. To eliminate this overhead, we perform most of the linking on the server side and replace symbolic names with integers. Code is transferred to the client in the form of pre-linked basic blocks instead of methods. The basic block granularity is beneficial because many methods contain code that is rarely executed, e.g., code for exception handling.

In a previous proof-of-concept implementation, we showed that SlimVM's approach is feasible [25]. Our previous implementation is based on the KVM [21], which does not support the full Java specification. Code is transferred from the server using method granularity, although some basic blocks are removed using static heuristics. This paper presents an implementation based on JamVM [15]. We now transfer code at the basic block granularity and support advanced features of Java like exception handling and the Java Native Interface.

Our measurements show that code size reduction, prelinking, and dynamic code loading of Java applications can reduce the memory footprint of a Java virtual machine by up to 70%. In summary, this paper contributes the following:

- We present a client-server architecture for mobile devices where all application and library code resides on a network server and is transferred to the client on demand at the basic block granularity.
- We show how advanced features of Java like exception handling, the Java Native Interface, and dynamic class loading are supported by SlimVM.

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
http://doi.acm.org/10.1145/1596655.1596678

- We evaluate our implementation and show the impact on memory footprint and network transfer.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of the overall architecture of SlimVM's approach. Section 3 presents how the server preprocesses code before it is transferred to the client on demand. Section 4 describes the modifications of the client virtual machine. Section 5 evaluates our implementation. Section 6 presents related work, and Section 7 summarizes and concludes our contributions.

2. SYSTEM STRUCTURE

The implementation of SlimVM is based on the open source Java virtual machine JamVM [15], a JVM designed for small memory footprint. The size of the stripped executable on PowerPC is approximately 220 KB and on Intel approximately 200 KB. It has a highly optimized interpreter for bytecode execution and uses a mark-and-sweep garbage collector. Unlike other lightweight JVMs, it supports the full Java specification including object finalization, the Java Native Interface, and reflection. It relies on the GNU Classpath Java library [10], which consists of about 7300 classes with a total size of about 15 MB.

The principal approach of SlimVM is the idea that a Java virtual machine can be divided into two: server and client. The server is in charge of analyzing the whole program prior to execution on the mobile device. The client is stripped-down to the functionality of receiving and executing modified bytecode instructions. This structure allows all application and library code to remain on the server, and only the parts of the code that are going to be executed are transferred to the client at run time.

A compiler such as javac translates a Java program into a hardware and operating system independent binary format, known as the *class file format* [14]. As illustrated in Figure 1, the *slim compiler* processes every class file needed by the Java application to be executed. It translates the class into a new *slim class format* that does not include any actual bytecode instructions. All bytecode instructions are kept separately in the form of basic blocks. Whenever possible basic blocks are pre-linked. Symbolic references to other methods and fields are already resolved, i.e., the resolution for bytecode instructions that refer into the constant pool are done on the server before sending information to the client.

In our implementation, each class and each method are represented by a unique integer number, and field names are replaced with memory offsets. Calls for statically bound methods reference the appropriate method number directly. Only dynamically bound methods require a method search at run time. However, this search is also based on the method number and not on the method name.

As illustrated in Figure 2, client and server are connected via a network connection. The server offers a number of callback functions in order to support dynamic loading of classes in the *slim class format*, bytecode instructions in form of basic blocks, and meta information. If the connection is lost or cannot be established SlimVM shuts down. In this case no class information and no bytecode instructions can be requested and therefore no Java program can be executed.

Due to the fact that class file information and bytecode instructions are kept separately on the server, it is possible

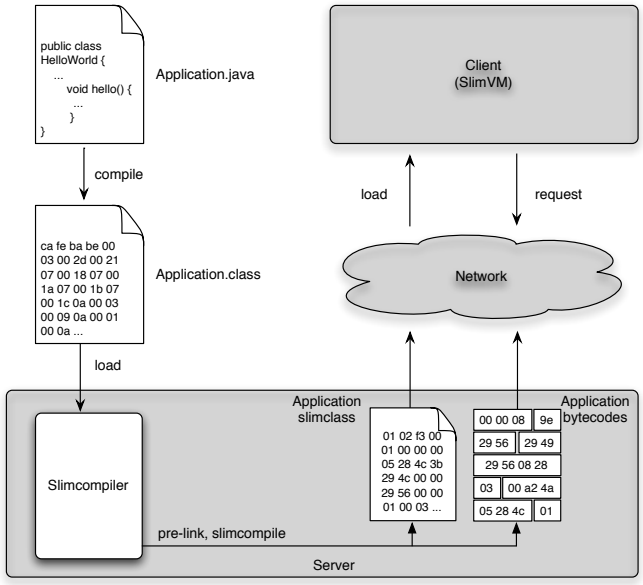

Figure 1: System structure of SlimVM

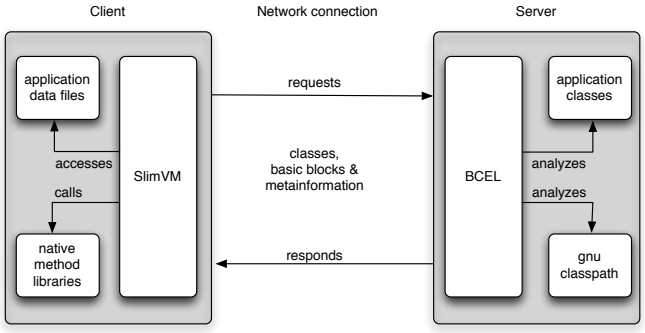

Figure 2: Client-server architecture

that only that parts of the code which are going to be executed are transferred to the mobile device. The analysis is not feasible for native code libraries that are invoked using the Java Native Interface (JNI), so all such libraries are kept on the client. From the point of view of the running Java application, the network connection is not visible, so the application executes as if it was fully loaded on the mobile device. Therefore, all application data files are stored on the client as well.

3. SLIM COMPILER

The slim compiler translates Java class files into the new *slim class format*. It uses the *Byte Code Engineering Library* BCEL [6], a library that 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 as objects. We use these objects to modify and to generate new bytecode.

We modify the following sections of a standard Java class file:

- A *Header* that contains the version of the used class format, several flags like access rights and class attributes.
- *Fields* define the names and types of instance and class fields.
- The *Constant Pool* is used to store constants and symbolic names
- *Methods* contain the method names and signatures together with bytecodes and other method attributes.

The constant pool 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. Although one constant can be referenced by multiple bytecodes to avoid duplicate entries, the constant pool contains the largest portion of an average class file, approximately 60%, whereas the bytecode instructions themselves just make up 12% of an average class file [1].

As memory consumption and bandwidth are a bottleneck for connected embedded systems, it is crucial to strip unnecessary information of a Java class file. Therefore we replace method names and class names, which are currently represented as string constants in the constant pool, by numeric identifiers and modify the corresponding instructions. Hence, it is possible to drop the whole constant pool except string constants, for example the string `Hello World!` of a hello world program. For this, the slim compiler uses two major data structures:

- The *class mapper* is a lookup table that maps every class name (including its fully qualified package name) to a numeric identifier.
- The *method mapper* is a lookup table that maps every method name (including its signature) to a numeric identifier.

Consequently, there is a unique number for each fully qualified class name, like java.lang.String, and a unique number for each method name, like substring(II)Ljava/lang/String for example. The substring method takes two int values (the start index and the end index) as a parameter and returns a new string that is substring of the given string.

Arrays are represented as symbolic names by one or more leading [followed by the name of the class. Since class names are no longer represented in the form of string constants, we use the most significant 8 bits of the classid to specify the nested array depth. As illustrated in Figure 3, the remaining 24 bits indicate the actual class number. This leads to a limitation for arrays with a nested depth larger than 255, but for all practical purposes we consider this limitation as negligible.

Beginning with system classes like java.lang.Object, java.lang.Class, or java.lang.String and methods like getSystemClassLoader(), the slim compiler reads and analyzes all class files needed during the startup phase of SlimVM. These classes and methods are loaded in the corresponding mapper first because this information is always needed during startup. Our implementation ensures that these classes and

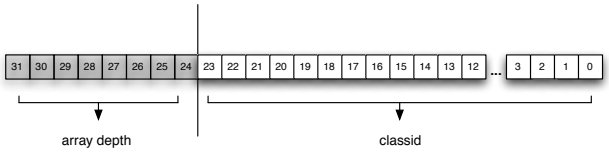

Figure 3: Representation of the array depth

methods always have the same identifier. Also primitive types, like for example int or char, get a distinct numeric identifier assigned during this phase.

Then, starting with the class file containing the main() method, the slim compiler analyzes recursively all dependencies of this class. Every new class and method is mapped to a numeric identifier. The offsets for all static fields and object fields of a class are calculated and stored for every class. This is necessary to replace the field resolution within the constant pool. The static size and object size for each class, which is needed by the Java virtual machine for allocating memory, is calculated during this phase.

Thereafter, the slim compiler starts analyzing the actual bytecode instructions which are segmented into basic blocks. All basic blocks are visited and instructions which need to be updated are modified. Finally, the collected information (except of all bytecode instructions which remain on the network server) is written out in the new *slim class format*.

3.1 Basic Blocks

A *basic block* is a sequence of instructions with only one entry and one exit point. That means that no jump instructions and also no destinations of jump instructions are included within a basic block. Only if a bytecode throws an exception, a basic block can be exited before its end. By transferring code at the basic block granularity, it is nearly guaranteed that every transferred instruction is actually executed.

We split all methods into basic blocks and number them in ascending order. The original Java semantics for branch instructions where a relative offset defines the jump target does not work any more. We change all relative offsets to absolute basic block numbers.

3.2 Modified Instructions

The instruction set of the Java virtual machine consists of one-byte opcodes followed by zero or more operands. Not all of the possible 256 instructions are used. The instruction set currently consists of 212 opcodes, 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 onto the operand stack of a Java virtual machine frame and vice versa.
- **Arithmetic Instructions:** These instructions usually compute a result of two values on the operand stack and push the result back on the operand stack.
- **Type Conversion Instructions:** These instructions allow the conversion between Java virtual machine numeric types.

- **Object Creation and Manipulation:** These instructions are used to create objects as well to put and to get values from that object.
- **Operand Stack Management Instructions:** Some instructions within the Java virtual machine directly manipulate the operand stack.
- **Control Transfer Instructions:** Branch instructions could cause the Java virtual machine to continue execution with an instruction other than the one following the branch instruction.
- **Exceptions:** An exception can be thrown either using a bytecode or by the Java virtual machine if an abnormal condition is detected.
- **Method Invocation and Return Instructions:** There are four instructions that invoke methods within the Java virtual machine:
  1. InvokeVirtual call instance methods of an object.
  2. InvokerInterface are used for methods defined in an interface.
  3. InvokeSpecial invokes constructors and other statically bound instance methods.
  4. InvokeStatic call static class methods.
  5. Return instructions are distinguished by their type.

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 scheme. The actual Java bytecode indexes into the constant pool. The constant at this index is either a primitive type which holds for example a field, method, or a classname, or it refers to other entries in the constant pool holding further information in order to resolve the correct field, method, class, etc.

Since all class files needed for execution of a Java program are analyzed on the server prior to execution on the mobile device and since we dropped the whole constant pool, we manipulated all instructions with a reference in the runtime constant pool.

As illustrated in Figure 4, the invokespecial opcode is no longer followed by an index into the runtime constant pool. The manipulated invokespecial opcode is followed by a numeric identifier for a class, a number for the corresponding method, and the number of the corresponding basic block.

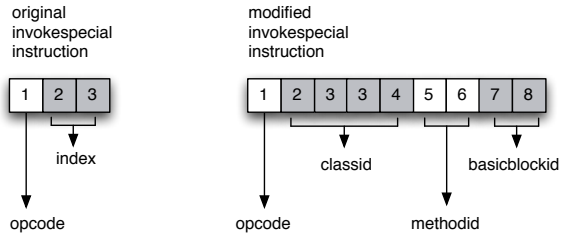

Figure 4: Invokespecial comparison

Resolving virtual methods is not based on a virtual method table like in a common JVM. In our implementation the invokevirtual instruction is followed by the class identifier, the method identifier and the basic block number of the base class. In order to get the number of arguments, the base class is resolved. Since we do know the arguments to be received by the virtual method, we are able to take the this pointer from the stack which lies beneath the arguments. Since each method has a unique identifier and since every method stores the unique numeric identifier of the superclass, we are able to resolve the correct method. Starting in the class of the dispatching object we traverse the superclass chain until a match is found. Invoking an interface method follows a similar scheme where the interface method of the base class is resolved first and then the method implementing that interface.

As illustrated in Figure 5, the original getfield opcode is followed by 2 bytes indicating the index into the runtime constant pool where the value to be pushed onto the operand stack is stored. The modified opcode is followed by an 1 byte identifier which indicates whether the field is 8 bytes or 4 bytes in size and therefore whether 8 bytes (long, double) or 4 bytes (int) are pushed onto the operand stack. Another 2 bytes indicate the offset within the object. Thus, pre-linking increases the size of every manipulated instruction, but makes the use of the constant pool redundant.

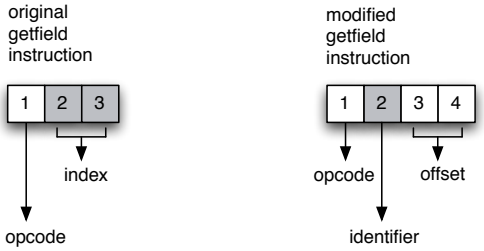

Figure 5: Getfield comparison

3.3 Slim Class Format

Equally to an original class file, the slim class file represents exactly one class or interface and consists of a stream of bytes where multibyte data items are stored in big-endian order. Like an original class file, the slim file includes all information required to allocate memory on the heap like the static size and the object size of a class. Furthermore, the slim class holds flags indicating whether a class has a main method, a static constructor or is abstract. A slim class also provides information how many interfaces are implemented. In contrast to the original file format only a reduced constant pool is present in the file including constants for output purposes. For every method we store the unique method identifier, the number of basic blocks within that method and a flag in case the method is native. The biggest difference is that no bytecode instructions are present in a slim file. All bytecodes are kept separately in form of basic blocks on the server which can be requested individually.

4. CLIENT-SIDE VM MODIFICATIONS

In order to support dynamic code loading, interpret modified instructions and to load classes in the new format, we customized the Java virtual machine JamVM. As illustrated in Figure 6, we modified parts of JamVM in order to make it feasible for our approach. Similar to a conventional

JVM, where the *Class Loader* is responsible for the dynamic loading and thereby creating the internal data structure of classes, the *Class Loader* in SlimVM is in charge of loading classes but with the difference that the class information is requested over a *Network client*. This *Network client* is not only responsible for requesting class information, but also for requesting bytecode in form of basic blocks and meta information.

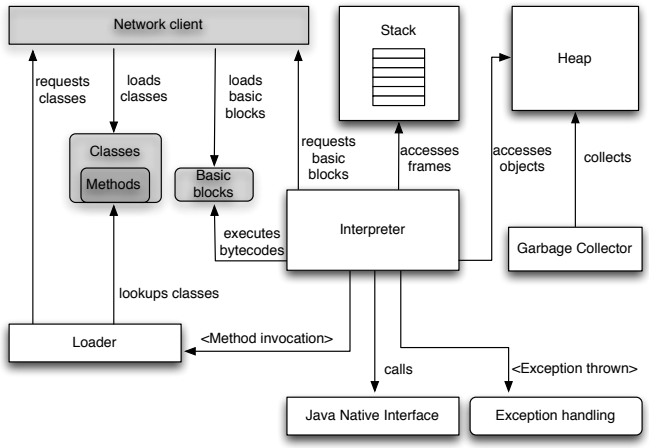

Figure 6: SlimVM architecture

Whenever execution of a new application is started, the client establishes a network connection to the server and starts with the initialization process. Once 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. In JamVM the *Linker* checks whether the class of an invoked method has been loaded so far or not. As linking in SlimVM is done on the server prior to execution on the client, the functionality of the *Linker* is limited to a simple *Loader*. Every time a new method is invoked, the *Loader* checks whether the class information of this method has been transferred to the mobile device so far or not. Once the class information is loaded in SlimVM, the *Interpreter* can directly request basic blocks over the network client. All class information and basic blocks requested once are stored on the mobile device which counteracts the overhead of transferring the same information over and over again. The modifications for SlimVM disregard garbage collecting.

While this approach usually works, there are four parts in the virtual machine that need additional support: dynamic class loading, reflection, exception handling, and the Java Native Interface.

4.1 Dynamic Class Loading

If the client wants to load a class that was not part of the original class subset, we have to load and analyze it during runtime. The slim compiler takes the name of the class and loads it. Equal to the initial phase of slimcompiling, all classes referenced by an invoke method in this class are also analyzed, mapped in the *classmapper* and compiled into the *slim class format*.

4.2 Reflection

By using reflection a program is able to to observe and to modify its own structure and behavior. Reflection gives access to class information, allows to read and write fields and even call methods of a class selected during run time. Since we change the representation of classes, methods and fields, we also have to adapt the way that reflection is handled within the VM.

For example, the reflection API is used to retrieve all constructors of a class using the method getConstructors(). This function returns an array of constructor objects. In our system, several flags define additional information for every method. These flags indicate access control or among others if a method is a constructor. For this example, we traverse all methods of the corresponding class and choose each method with the constructor flag set. In order to allocate enough memory for each object, we need the signature to calculate the size. The signature of the constructor public Foo(String mystring, int myint) is (Ljava/lang/String;I)V. We only use a short version of the signature that indicates whether the signature represents a class, an array or a primitive type. In our example, the signature is reduced to (L;I)V in SlimVM's approach. Retrieving fields with the API function getFields() for example is processed in a similar way. Also flags are stored for every method indicating the size and the access rights of every field.

Another example is retrieving the name of a class using reflection. Since every class stores its own unique class identifier it is possible to call back to the server with this number and retrieve the corresponding class name.

4.3 Exception Handling

Usually exceptions in the Java virtual machine are handled with assistance of an exception table. Whenever an exception arises, the JVM performs a lookup into a table that stores ranges for exceptions and sets the program counter to the corresponding target where the exception code is present.

To support exception handling in SlimVM, we mapped every modified instruction to its original instruction. This mapping table is stored on the server. Whenever an exception occurs in the Java virtual machine, the client calls back to the server giving the numerical identifier of class and method, the number of the basic block and the offset within that basic block where the exception occurred. With this information the server is able to calculate back the original address where the exception occurred. Taking that address a lookup in the original exception table is performed where the offset to the exception code is stored. With this offset a lookup in the mapping table is performed where the corresponding basic block number of the catch block is stored. This data is sent back to the client and the corresponding basic block can be requested.

4.4 Java Native Interface

The *Java Native Interface* [13] is a native programming interface that allows Java code to interoperate with applications (hardware and operating system specific programs) and libraries written in other languages, such as C, C++, and assembly. Many basic Java library classes depend on the JNI to provide functionality such as I/O file reading.

In the approach of SlimVM, all native functions are present on the mobile device. As we changed the representation of classes and methods, we offer six callback functions in order

to resolve meta information. These functions offer a way to resolve method ids to method names, class ids to class names and vice versa.

For example, a class should be loaded through the native function forName() which takes the name of the class to be loaded from the top of the stack. As illustrated in Figure 7, the client calls back to the server giving the name of the class to be loaded. The server takes that name and performs a lookup in the mapping table to resolve the correct id of that class. In case the requested class has not been analyzed on the server so far, it is now loaded. The process of dynamic class loading is described in more detail in Section 4.1.

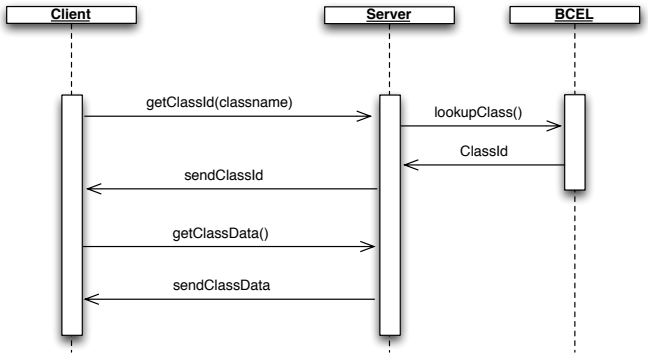

Figure 7: Resolving process of JNI meta information

5. EVALUATION

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

The test environment consists of a client and a server running on two separated systems which are connected via a 2.4 GHz WLAN. On the server side we use BCEL version 5.2 for analyzing and modifying Java class files. The actual server functionality is integrated in 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. 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.

We use a number of test programs and some selected benchmarks, tested on SlimVM and JamVM, to be able to compare the gathered results. We use a program with an empty main method, which is meant to pinpoint the startup overhead of SlimVM, a simple *hello world* application and a program which calculates the factorial of a given number. In order to be able to make some standardized and comparable statements we use some selected benchmarks: The Linpack [11] benchmark, which performs numerical linear algebra, like vector and matrix operations. The Scimark 2 [17] benchmark, which measures numerous computational kernels and summarizes the score in approximate Mflop/s. The FloatingPointCheck of Specjvm98 [20], which measures a number of floating point operations. The ArithBench and

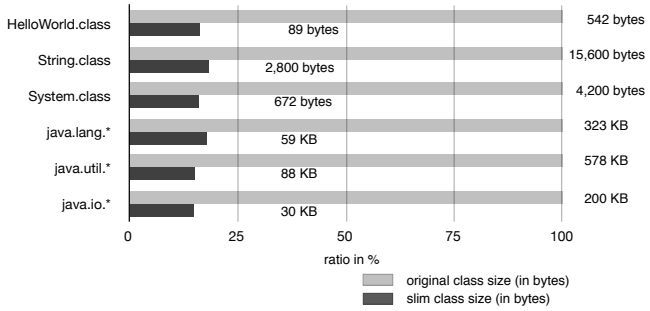

Figure 8: Comparison of the class size

LoopBench of JavaGrande [3], which measure arithmetical and loop operations.

5.1 Class Size Reduction

Stripping unnecessary information of a class file, using a different way for resolution of classes, methods and fields and dropping the constant pool makes it possible to shrink the size of a Java class for transferring runtime critical information to the client. The fact, that all bytecode instructions are swapped out to the server reduces the size of the *slim class format* in addition. However, as illustrated in Figure 8, we are able to reduce the size of Java classes by about 85%.

Dropping the constant pool of the class representation and pre-linking of basic blocks leads to an overhead for instructions in terms of multiple transfers of identical constants. For example, an integer constant is pushed on the stack five times in a while loop using the ldc instruction. In this case, one and the same integer constant is transferred to the client five times. We checked the most common classes like java.lang.Object, java.lang.Class, java.lang.String, etc. for the multiple use of integer constants and double constants. Our measurements show, that appearance of ldc instructions in these classes does not exceed four. Only once during our measurements, the same constant was shared by two load constant instructions. Thus, we consider the overhead of transferring the constant value as an operand of *ldc* instruction as negligible.

5.2 Saved Memory Space

One of the most interesting facts we discovered is, that a common JVM loads a high amount of a programs bytecode into memory even though few bytecode suffice for the correct execution of the program.

For a simple *hello world* program for example, we analyze 1,268 class files on the server whereas only 195 classes are loaded in SlimVM (as well as in JamVM) for execution of that program. The large number of analyzed classes is due to the fact that all classes referenced by an invoke instruction are loaded and analyzed recursively on the server. Furthermore, we analyze 38,800 basic blocks whereas only 1,000 basic blocks with a size of 48 KB are loaded. In contrast to JamVM, which loads 98 KB of bytecode instructions. As illustrated in Figure 9, this demonstrates, that about 50% of the bytecode instructions loaded by a virtual machine are not executed and therefore not needed for the correct execution of that program.

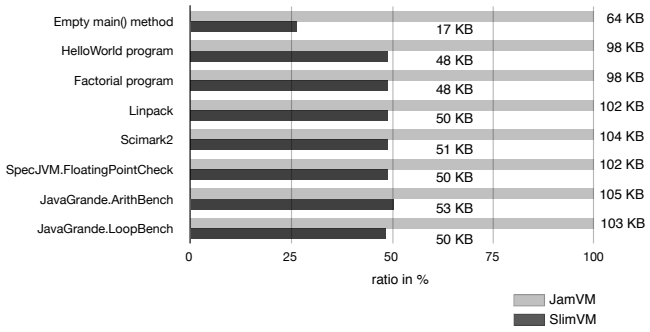

Figure 9: Space savings for lazy loading of bytecode instructions

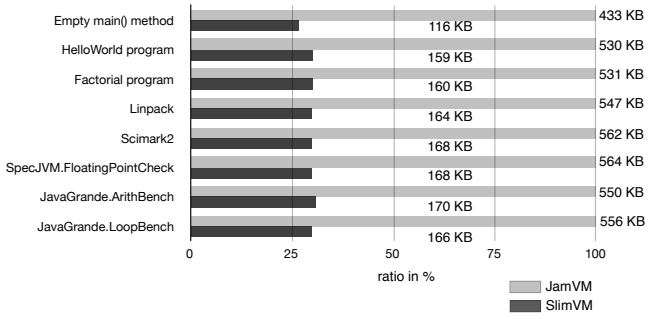

Figure 10: Space savings for lazy loading of classes including bytecode instructions

Measured on the full GNU classpath library, which is about 15 MB in size, we reduce the code footprint for a simple *hello world* program to 159 KB, which is a reduction of more than 99%. Subtracting 116 KB which are needed for initialisation purpose of the virtual machine, we can execute that program by loading only 43 KB.

Figure 10 illustrates the ratio between bytecode loaded by SlimVM compared to memory loaded by JamVM. About 70% of the memory loaded by a common virtual machine are not needed to execute that program. This pinpoints the overhead of redundant code loaded by a common JVM.

5.3 Runtime Distribution of Code Requested

Figure 11 shows the runtime distribution of code requested from the server for a *hello world* program. As illustrated, we record a continuous rise of code transfers for execution of the program. This is due to the fact that execution of that program uses different bytecodes and therefore they have to be requested and transferred from the server.

Figure 12 illustrates the runtime distribution of code requested for the *FloatingPoint* benchmark of SPECjvm98. This benchmark measures a number of floating point operations running through a for loop for one million times. Therefore code is only requested till the loop starts. Once the loop is started, no code has to be requested from the server because all bytecode transferred once, is stored on the client.

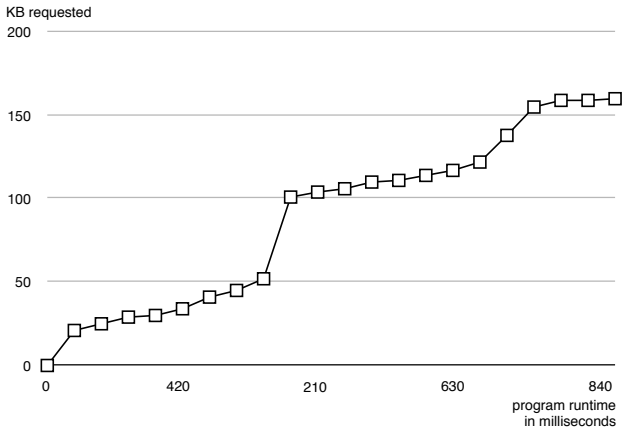

Figure 11: Runtime distribution of code requested of a HelloWorld program

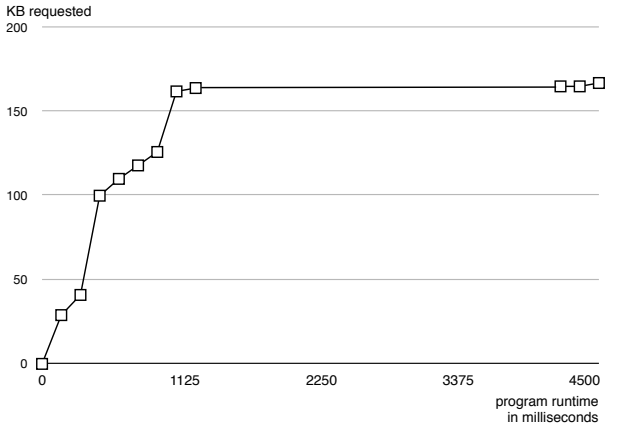

Figure 12: Runtime distribution of code requested of the FloatingPointCheck benchmark of SPECjvm98

5.4 Network Transfers

In order to execute a Java program on the client we support a number of callback functions to request class information in the new format, bytecode in form of basic blocks and functions to request meta information. As mentioned in Table 1 more than 1,500 callbacks are performed for the execution of a *hello world* program. The number of 12,000 callbacks for execution of the Scimark 2 benchmark is due to the fact that the native function arraycopy and the native function sin are called over and over. Every time a native function is invoked, the client calls back to the server in order to resolve a method number to a method name in order to call the correct native function.

Table 2 gives an overview of the transferred classes and basic blocks. For the Linpack benchmark for example, we analyse 39,000 basic blocks and 1,269 classes on the server whereas only 1,225 basic blocks and 206 classes are transferred to the client.

5.5 Effect on Performance

	callbacks
Empty main() method	1,145
HelloWorld program	1,533
Factorial program	1,537
Linpack	1,698
Scimark2	12,227
SpecJVM.FloatingPointCheck	1,907
JavaGrande.ArithBench	2,877
JavaGrande.LoopBench	1,966

Table 1: Number of callbacks to the server

	classes transf.	BB transf.
Empty main() method	150	796
HelloWorld program	195	1,044
Factorial program	195	1,048
Linpack	198	1,166
Scimark2	206	1,225
SpecJVM.FloatingPointCheck	205	1,122
JavaGrande.ArithBench	204	1,218
JavaGrande.LoopBench	203	1,131

Table 2: Classes and basic blocks transferred

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

Figure 13 and Figure 14 show two benchmarks of JavaGrande. The ArithBench calculates the number of additions, multiplications and divisions that can be executed over a certain period of time which is displayed in thousands of adds, multiplies and divides. The LoopBench measures the performance of looping constructs which is denoted in thousand iterations. As illustrated, SlimVM has a performance decrease in contrast to JamVM, which is furthermore caused by basic block jumps. A conventional JVM is able to make jumps for and back in the bytecode whereas SlimVM has to perform a lookup every time a basic block is going to be executed. In case the basic block has not been transferred to the mobile device, it has to be requested from the server which takes additional time which decreases the performance.

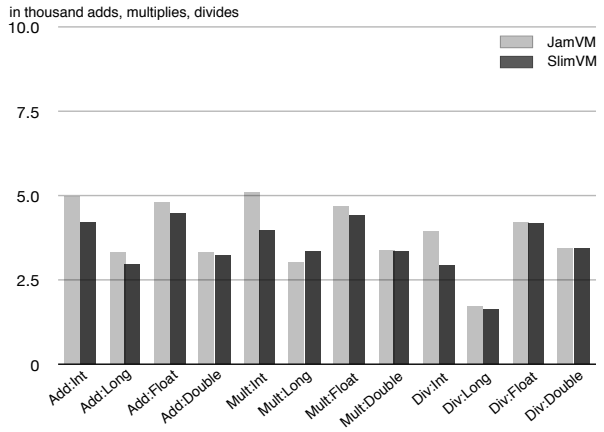

Figure 13: ArithBench section of JavaGrande

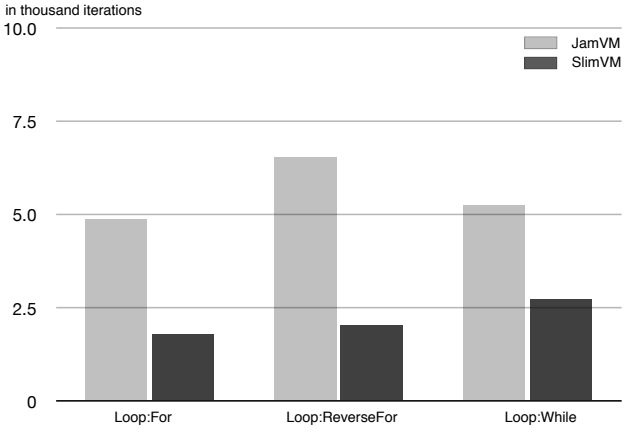

Figure 14: LoopBench section of JavaGrande

6. 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. Our work does not only contribute to shrink the size of Java applications, but also to examine a more efficient intermediate representation for transferring Java applications.

The standard means of packaging Java class files for distribution and storage is Sun's JAR [22] file format, which aggregates many Java class files as a single unit using the well known zip [16] compression mechanism. JAR reduces the size of Java class files by almost 50% and can be executed by any Java virtual machine.

Bradley et al. present a format called Jazz [2]. Similar to JAR, Jazz bundles a number of class files together and compresses them. They eliminate redundant constant pool entries and combine the constant pools of all compressed classes so that method names, signatures, integer constants only appear once in the constant pool, no matter how many classes make use of it. This reduces the data to 25% of its original size.

Pugh presents a custom compressed format for collections of Java class files called the wire-code format [18]. A compressor transforms JAR files into the wire-code format and reduces the size by sharing information across class files. Furthermore, Pugh encodes method types as an array of classes containing the return and argument types instead of using strings. Finally, generic attributes are eliminated by using additional flags in the access flags section. The wire-code format is typically 50% - 80% smaller than the original JAR file using the compression algorithm gzip [7].

Rayside et al. present an effective solution for the conflicting requirements of code size reduction and execution per-

formance called compact Java binaries [19]. To reduce the size of the constant pool they 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 they separate opcodes from operands applying different techniques like using the Huffman algorithm in order to reduce the number of bits required to represent the most frequent opcodes. Rayside et al. modified the constantpool and the code attribute of class files in a way that their evaluation shows a typical size reduction of 25% for class files and 50% for JAR files.

Clausen et al. present in their work *Java Bytecode Compression for Low-End Embedded Systems* [5] that factorization of common Java bytecode instructions can reduce the memory footprint, on average, to 85% of its original size with a minimum time penalty.

Latendresse and Feeley [12] 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. Their compression factors highly depend on the original bytecode, but typically vary between 30% and 60%.

Yang [26] reduces the overall memory footprint of the constant pool to about 87% of their original size by performing pre-resolution. Close to our work, he resolves all references to other class definitions in the constant pool but does not support dynamic binding.

Most former work on dead code elimination is focused on improving the efficiency of a program. Butts [4] has shown that approximately 3% to 16% dead or unreachable code exists in programs.

Franz and Kistler present slim binaries [9], a compact platform-independent program representation, which is designed to be translated into binary code by an optimized JIT compiler. Based on the fact that different parts of a program are often similar to each other, these similarities are exploited by using an predictive compression algorithm which allows the encoding of recurrent subexpressions in a program space efficiently. Their approach can reduce the size of a complete application by factor of three.

Ernst et al. present a compressed executable called BRISC [8], 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. They scan the input program over and over and add frequently occurred instructions to a dictionary and calculate the program size reduction if the candidate were added to the dictionary minus the number of bytes needed to represent the instruction pattern in the dictionary. Once a dictionary is created, the dictionary followed by the modified input program are written to the BRISC file. As the results show, BRISC is a good mobile program representation choice which is competitive with gzip in size.

Titzer et al. present a dynamic VM where all code, data and VM features are packed into a binary image [24]. Feature analysis detects unused code and data. Furthermore, Java features are also subject to be removed from the VM if the application does not use them. The memory footprint for non-heap memory allocation can be reduced by up to 75%.

7. CONCLUSION AND OUTLOOK

We have presented a Java virtual machine for persistent connected embedded devices. Off-target loading and verification allows us to leave all application and library class files on the server. The client part of our VM requests only information that is needed for executing a program.

Our results show that the transfer size of applications is reduced by up to 70% in comparison to lazy class loading with standard Java class files.

In comparison to previous work, we completely switch to basic block granularity. No dispensable instruction is sent to the mobile device at all. We are also supporting now Java features such as the standard library, JNI, reflection, and complete exception handling.

As far as future work is concerned, we are particularly interested in adding dynamic code request analysis on the server side. We want to keep track of all basic blocks that are requested from the client and use this information to calculate the likelihood of following basic blocks. Depending on our heuristics and parameters like network-bandwidth or memory constraints on the client side we can combine basic blocks that are transfered to the client and therefore reduce the number of requests and execution time.

We are also interested in exploring on-target verifiable code compaction. Our current framework relies on a trusted communication channel between the loader and the target device. We believe it is possible to make our wire format verifiable by the target device without adding a significant meta information overhead. This would relax the requirements on the communication channel with the off-target loader.

Acknowledgment

This research effort is partially funded by the National Science Foundation (NSF) under grant CNS-0615443. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation (NSF), or any other agency of the U.S. Government.

8. REFERENCES

[1] D. N. Antonioli and M. Pilz. Analysis of the Java class file format. Technical report, Department of Computer Science, University of Zurich, 1998.

[2] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 294–302. IBM Press, 1998.

[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[4] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *ASPLOS-X: Proceedings*

of the 10th international conference on Architectural support for programming languages and operating systems, pages 199–210, New York, NY, USA, 2002. ACM.

[5] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[6] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, 2001.

[7] P. Deutsch. *GZIP File Format Specification Version 4.3*. http://www.ietf.org/rfc/rfc1952.txt.

[8] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco. Code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 358–365. ACM Press, 1997.

[9] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[10] *GNU Classpath*, 2009. http://www.gnu.org/software/classpath/.

[11] M. Jeckle. *Linpack Benchmark – Java Version*, 2004. http://www.jeckle.de/freeStuff/jLinpack/.

[12] M. Latendresse and M. Feeley. Generation of fast interpreters for Huffman compressed bytecode. In *Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators*, pages 32–40. ACM Press, 2003.

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

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. The Java Series. Addison-Wesley, 1999.

[15] R. Lougher. *JamVM*, 2009. http://jamvm.sourceforge.net/.

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

[17] R. Pozo and B. Miller. *SciMark 2.0*, 1999. http://math.nist.gov/scimark2/.

[18] W. Pugh. Compressing Java class files. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258. ACM Press, 1999.

[19] D. Rayside, E. Mamas, and E. Hons. Compact Java binaries for embedded systems. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 1–14. IBM Press, 1999.

[20] Standard Performance Evaluation Corporation. *The SPECjvm98 Benchmarks*, 1998. http://www.spec.org/jvm98/.

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

[22] Sun Microsystems, Inc. *JAR File Specification*, 2003. http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html.

[23] Sun Microsystems, Inc. *Java ME at a Glance*, 2009. http://java.sun.com/javame/.

[24] B. L. Titzer, J. Auerbach, D. F. Bacon, and J. Palsberg. The ExoVM system for automatic VM and application reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 352–362. ACM press, 2007.

[25] G. Wagner, A. Gal, and M. Franz. SlimVM: Optimistic partial program loading for connected embedded Java virtual machines. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java*, pages 117–126. ACM Press, 2008.

[26] Y.-S. Yang, M.-S. Jin, S.-I. Jun, and M.-S. Jung. A study on an efficient pre-resolution method for embedded java system. pages 20–24, July 2004.