Science of Computer Programming 76 (2011) 1037–1053

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

"Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading

Gregor Wagner *, Andreas Gal, Michael Franz

University of California, Irvine, CA 92697-3425, USA

ARTICLE INFO

Article history:
Received 20 July 2009
Received in revised form 19 October 2009
Accepted 2 December 2009
Available online 29 April 2010

Keywords:
Java virtual machine
Just-in-time compilation
Embedded connected devices
Cold code removal

ABSTRACT

Embedded systems provide limited storage capacity. This limitation conflicts with the demands of modern virtual machine platforms, which require large amounts of library code to be present on each client device. These conflicting requirements are often resolved by providing specialized embedded versions of the standard libraries, but even these stripped down libraries consume significant resources.

We present a solution for "always connected" mobile devices based on a zero footprint client paradigm. In our approach, all code resides on a remote server. Only those parts of applications and libraries that are likely to be needed are transferred to the mobile client device. Since it is difficult to predict statically which library parts will be needed at run time, we combine static analysis, opportunistic off-target linking and lazy code loading to transfer code with a high likelihood of execution ahead of time while the other code, such as exception code, remains on the server and is transferred only on demand. This allows us to perform not only dead code elimination, but also aggressive elimination of unused code.

The granularity of our approach is flexible from class files all the way down to individual basic blocks. Our method achieves total code size reductions of up to 95%.

1. Introduction

Java virtual machines are used across the full spectrum of computers, from servers to personal computers, to mobile and special-purpose devices. They are popular because of the "write once, run anywhere" benefit of the Java language. In the mobile space, two deployment scenarios have crystallized: The first is to optimize for performance using an unmodified Java virtual machine with only minor changes in the application code. The second is to minimize code size by providing new Java class file formats reducing the amount of data transferred over a network. Less data to transfer reduces not only the transfer time and latency, but also the memory requirements for the mobile device. Even on smart phones such as the iPhone, transfer size is still an issue due to the limited bandwidth of a modern cell network.

Sun's Connected Limited Device Configuration (CLDC) [20] standard targets these special mobile devices with restrictions for processor speed, display, and network connection. Everything is trimmed to be small and limited in memory and power. The Java Platform Micro Edition (Java ME) [18] integrates the CLDC standard and provides a development framework for mobile applications.

Our goal is to go further by exploring a zero footprint paradigm for connected embedded devices. All code, including application code and library code, reside on a remote network host. The mobile device contains no code at startup. The device requests and executes code as needed, and only the small subset of class files that will likely be required.

* Corresponding author.
E-mail addresses: wagnerg@uci.edu (G. Wagner), gal@uci.edu (A. Gal), franz@uci.edu (M. Franz).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.04.008

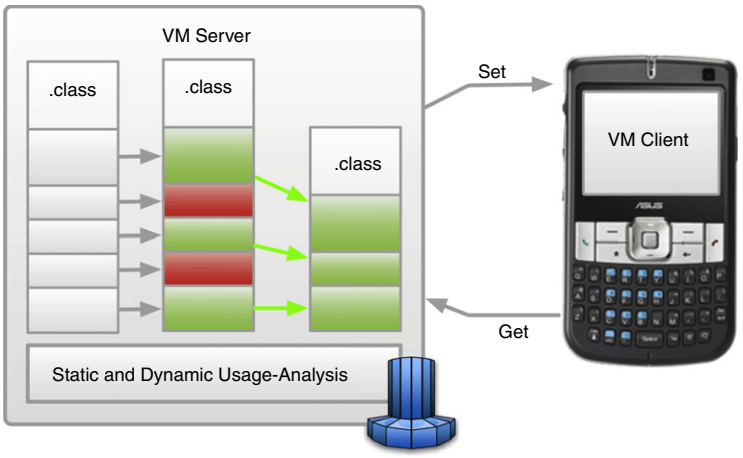

Fig. 1. *SlimVM overview.*

Code that is less likely to be executed such as exception code, remains on the remote host and is provided only on demand. The main challenge is that it is difficult to predict ahead of execution which parts of a library will actually be needed at run time. Many components of a library are referenced statically, but are never actually invoked at run time.

To find rarely executed code, we combine static analysis and lazy code loading in our new *SlimVM* approach. As can be seen in Fig. 1, our server-side analysis tool called SlimAnalyzer loads standard Java class files. We split all class files down to the granularity of basic blocks. Our approach not only deals with *dead* code elimination, but also removes parts of functions that are *unlikely* to be executed. Based on tunable settings (corresponding to a more or less optimistic approach), this slimmed pre-linked bytecode is shipped to the mobile device. If our heuristic fails and code is needed during execution that was marked as unlikely executed, we send a reload request to the server to continue execution.

In summary, this paper contributes the following:

- We present a client–server architecture for embedded mobile devices where all application and library code resides on a network server and is transferred to the client on demand.
- We use static analysis on the server to identify most likely needed code.
- We show that code size reduction, pre-linking and on-demand code provisioning can reduce application and library size by up to 95%.
- We show that the overall memory consumption of the virtual machine decreases up to 75%.

The remainder of this paper is structured as follows. The next section gives a brief introduction to the Java bytecode format and some of its peculiarities. Sections 3 and 4 discuss removal of data in the constant pool of Java class files and fine-grained remote and partial loading. Following a section presenting benchmarks, we summarize and conclude with related work in Section 6 and an outlook to future work.

2. The Java bytecode format

Java source files [9] are transformed into a machine-independent instruction format called Java bytecode by a compiler such as `javac`. Each compiled class file contains four main sections: constant pool, fields, methods, and attributes. Statistical analysis [1] indicates that 60% of an average class file is used by the constant pool, as it holds all constants and symbolic references to fields, methods, and classes. The bytecode part of the class file is only 12% of the total space.

The Java virtual machine uses a stack-based execution model. For example, the `getstatic` opcode, which pushes a reference to a static field onto the stack, is followed by an index into the constant pool. This special index is a constant field reference, which references the fully qualified name of the class in which the field resides, and the name and type of the field.

Fig. 2 (modified from [6]) shows some parts of this `HelloWorld` class :

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

In the corresponding compiled bytecode, instruction 1 pushes a field named `out` from the class `java.lang.System` onto the stack. It is an instance of the class `java.io.PrintStream`. Instruction 2 pushes a reference to the string constant *"Hello*

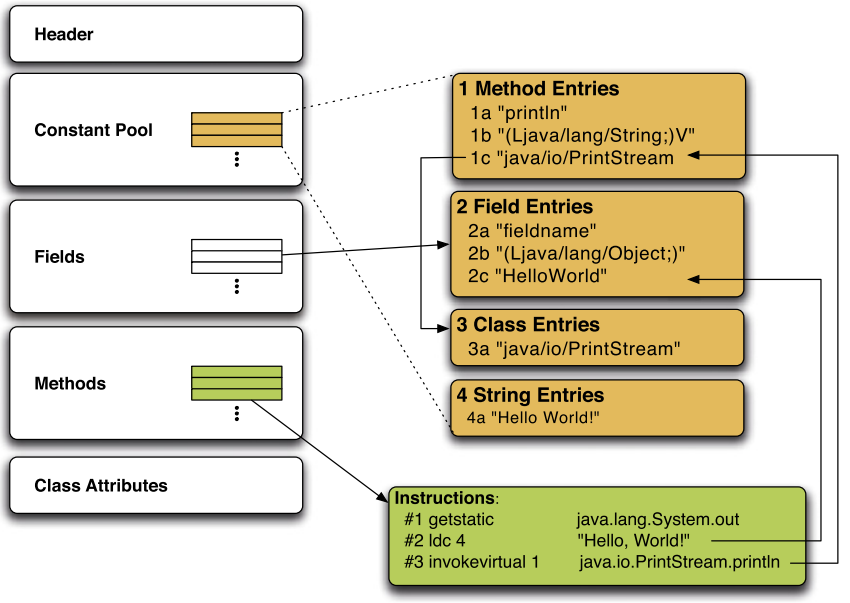

Fig. 2. Structure of a Java class file. In the instructions section there are bytecode instructions implementing the well-known `System.out.println("Hello World!")`. Executing these instructions results in lookups into the constant pool. In the constant pool entries part, there are some examples of what class, field, method and string entries in the constant pool look like. Method 1, for example, is identified by its name (1a) and its type (1b), which represents a parameter of the class `java.lang.String` and a return value of the type `void`. Finally, entry (1c) specifies the location of this method, namely in class `java.io.PrintStream`.

World" onto the stack. Finally, instruction 3 invokes the method `println` by referring to a constant method reference entry in the constant pool.

First the method `println` in class `java.io.PrintStream` with one string parameter and the return value `void` is searched. If this succeeds, the instruction pointer is set to the beginning of the bytecode of this method. During the execution of a bytecode sequence, many lookups like this must be performed, and symbolic references to classes, fields, and methods must be dynamically resolved.

A common optimization is to replace constant pool lookup instructions with special, already resolved instructions that are often termed *quick* or *fast* bytecodes. We perform this optimization as often as possible during our analysis process, as described in Section 3. Another typical optimization, resolution of virtual method dispatch, can be performed in the general case only when the specific class is known at run time. However, there are many examples where this resolution can already be performed at link time. Field accesses can be optimized in the same manner. As we show below in Section 6, removing complex lookups are one of the solutions that lead to improved, specialized Java class file formats for embedded devices.

A Java application normally requires all standard library functionality to be present at the start of execution. This means that even an embedded mobile device has to store *all* the library functionality locally in order to provide a suitable execution environment for all possible applications. The main problem that we tackle in our research is to predict ahead of time which parts of the application code and which libraries will actually be needed at run time, and to provide a method of recovering gracefully from a misprediction. This leads to a "zero footprint" environment in which *no library at all* is present a priori, and in which libraries are provided on demand in response to specific application requirements only.

3. SlimAnalyzer implementation and remote loading

The Java class file format is used on a broad range of machines with vastly different capabilities—from high-end applications on servers and workstations down to inherently limited embedded systems. While it is a waste of network and storage capacity to transfer any code to the client that is not subsequently executed, this has especially severe consequences in the case of devices with constrained resources. One approach of reducing the amount of data transferred is to use special-purpose file formats, which can reduce the size of whole applications dramatically. An alternative solution is to change the design of the whole execution environment, as for example done by Krintz et al. [11].

Our approach falls in this second category, as we focus on reducing the information content that is transferred to the client, rather than merely encoding the same content more densely. In order to transfer as few data items as possible, we have to perform two main steps during analysis:

- Identify all required information of the application and library.
- Extract and compress this subset from the application and library.

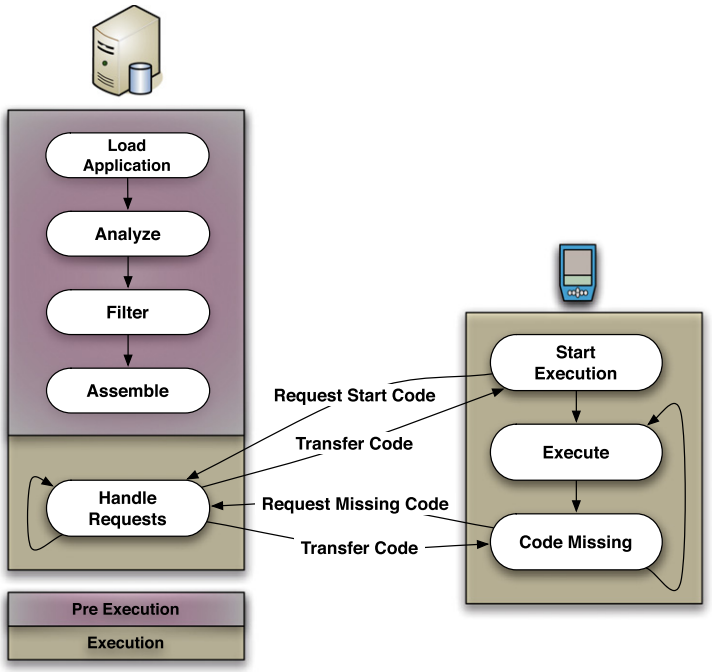

Fig. 3. Overall structure of our server–client architecture. Application and library class files are loaded by the SlimAnalyzer tool on the server. Using static analysis, we generate an optimistic subset of class files. Further optimization steps shrink and reorganize the resulting subset of class files. On the client, SlimVM starts executing using only the reduced code provided by the server. If the optimistically loaded code subset is incomplete, the missing code is later provided on demand.

The first step is performed on the network host as can be seen in Fig. 3. All application and library class files are loaded by the SlimAnalyzer. This tool is based on the "Bytecode Engineering Library" BCEL [6]. It is written in Java and runs inside the Java virtual machine. It represents classes, methods, fields and instructions as objects. This gives us the opportunity to remove parts of a Java class file easily.

Our first optimization step is to determine which application and library functionality is possibly needed at run time. We collect all class references from the constant pool beginning in the class where execution starts. This generates a subset of class files that we need to deal with for further optimization. A simple `HelloWorld` application, for example, requires a subset of 73 class files. The effectiveness of our optimization depends on a library's internal organization. If a library has high internal coupling, then it may not be possible to generate a significantly smaller subset of it.

Since 60% of an average class file is taken up by the constant pool, a big step to reduce transfer size is to change the representation of references. There is no need to name a class for example "java.lang.Object" when all we need is a unique identifier for the class. As a consequence, we change the representation from strings to numbers. Java.lang.Object becomes, for example, 0 and all references to this class point to class 0. We do the same with all strings that are responsible for lookups. These are names of methods, fields, classes and types.

We cannot change error messages or the "Hello World!" string that should appear on the screen in our example in Section 2. All strings that cannot be removed are stored in a separate string stream. A `ldc` instruction with a string identifier is now followed by the corresponding string stream offset. As can be seen in Fig. 4, the instruction is now followed by a type identifier and the offset in the string stream. This optimization leads to an optimization in the VM. The opcode `new` is followed by an index into the constant pool shown in Fig. 5. Since each class is represented by a number, we do not need a lookup into the constant pool. We just replace the constant pool index from opcode `new` with the class identifier number. As a result we can delete all class references and the corresponding class names from the constant pool.

Constant pool entries are not only used to perform class or method lookups, but also to support reflection. In the current version of our SlimVM implementation we do not support reflection. To support reflection, since most of the relevant information remains on the server, we only have to implement a constant pool entry request function.

Method invocations are more complex, but basically it is the same procedure. As shown in Fig. 5 each method is identified by its host class, name and type. The type of a method that takes a string as an argument and returns a string is encoded as

```
(Ljava.lang.String;)Ljava.lang.String;
```

Each modification of parameters or a return value needs a separate constant pool entry. Ideally, like the opcode `new`, we replace the index into the constant pool with the offset that this method has in the bytecode stream. This offset is calculated during the analysis process on the server host. If we cannot statically predict the called method, for example in the case of a virtual or interface method call, we replace the constant pool index with a unique identifier generated for this corresponding

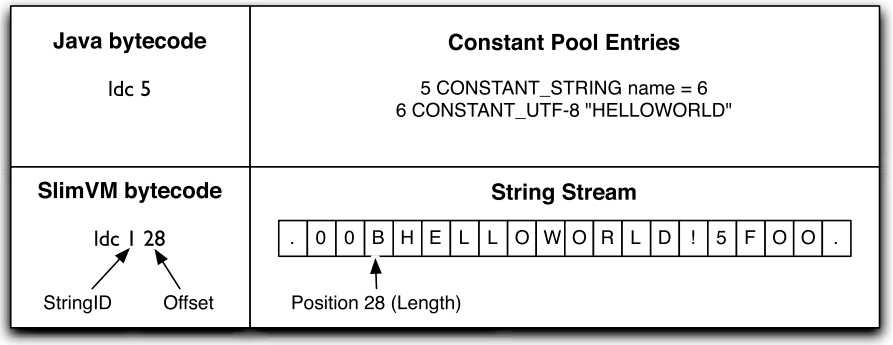

Fig. 4. All remaining strings from the constant pool are stored in a separate string stream. A ldc instruction is now followed by a type identifier and the offset for the string. In this example, the offset is 28. At position 28 in the string stream is the length of the current string followed by the string itself stored.

Java bytecode	Constant Pool Entries
new 5	5 ClassRef name = 23 6 String "java/langObject"
invokespecial 4	4 MethRef ClassRef = 5 NameAndType = 21 5 ClassRef name = 22 22 String "com/cldc/DefaultConv" 21 NameAndType name = 10 sig = 11 10 "isLowerCase" 11 String "(Integer)Integer"

Fig. 5. A more detailed example than Fig. 2. Opcode `new` points to a ClassReference entry. This ClassReference points to the class name which is represented by a string entry. More complex is a method lookup. This example is based on an `invokespecial` instruction. The MethodReference entry points to the corresponding ClassReference and NameAndTypeReference entries. These two entries point to the class name, method name and signature.

method. It consists of the new class name, new method name and new type. All of these identifiers are numbers as mentioned before. During execution, we have to perform a lookup in the specific object we get from the stack. There is a lookup table for each class in which the transferred or not transferred status and the corresponding offset is stored for each method.

The following list contains more detailed information about all instructions that are changed from the original bytecode definition:

`new`:
: The opcode `new` was originally followed by a short index referencing into the constant pool. This is changed to a short index that points to a class directly.

`anewarray`:
: The `anewarray` instruction is used to create a single dimension of an array of object references or part of a multidimensional array. It has a short argument that holds the reference into the constant pool. This argument is changed to a direct reference to the class file. Additionally, there is a single byte added that indicates whether the elements have a basic type such as int, float, etc. or a reference to another class. This is important since base classes are allocated in another way than reference classes in the virtual machine.

`multianewarray`:
: The multianewarray instruction creates a multidimensional array. The index into the constant pool was again changed to a direct reference to the class file.

`invokevirtual`:
: The invokevirtual instruction was originally followed by a short index that points to the constant pool. In order to perform method referencing without a complex constant pool lookup, we changed the representation of a method. The index to the constant pool is changed to a numeric identifier that represents the method name and another numeric identifier that represents the signature of the method. The bytecode is extended from 3 bytes to 5 bytes but we save many more bytes by removing the entry for the method from the constant pool.

`invokespecial, invokestatic`:
: These instructions are used to invoke instance initialization methods and class methods. In the original version, these instructions were followed by a short index into the constant pool. Like the invokevirtual instruction before, the index is changed to a numeric identifier for the name and a numeric identifier for the signature.

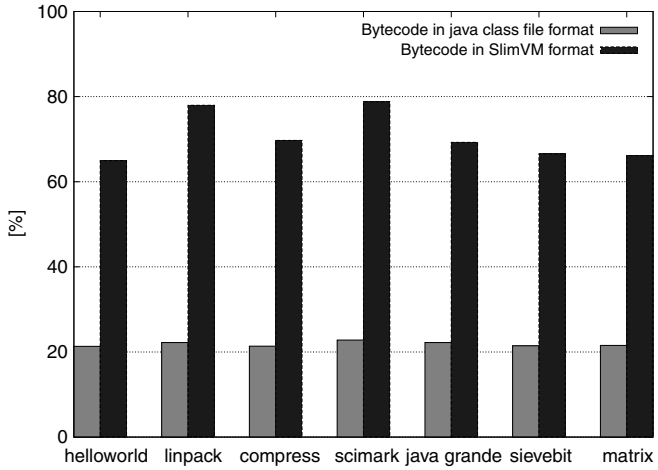

Fig. 6. Bytecode size vs. class file size of an application. In the standard Java class file format, there is a lot of meta-information such as the constant pool and the line number table. By removing this meta-information, we obtain the SlimVM ratio between file size and bytecode size.

`invokeinterface`:
: This instruction is used to invoke an interface method and has a short index that points to the constant pool and a single byte index that holds the value for the arguments of the method and one byte that is always zero. The argument index is used to get the class reference from the stack, since we do not know how many arguments an interface method has. This value is used to remove the right number of arguments of the stack in order to get the reference to the class.

`ldc, ldc_w, ldc2_w`:
: These instructions push constants from the constant pool onto the stack. Since we removed most entries of the constant pool, these instructions have to be adapted. A single byte is added in order to identify the constant that has to be loaded. All constants except string constants are stored right after the type identification byte in the bytecode stream. An integer is stored in the bytecode stream as follows: 0x121yyyy. 0x12 represents the ldc opcode, 1 represents the integer type and yyyy stands for any integer value. String constants are stored in a separate string array. If such a string is loaded, the value after the string identification type represents the offset in the string array. 0x1220010 means that a string has to be loaded from the internal string array with the offset 0x10.

Field accesses can be resolved to an absolute offset within an object. As a result, each field instruction references directly to the corresponding field and no time-consuming constant pool lookups are necessary. By now, with the exception of string constants, we have removed all constant pool references that are not responsible for class, interface, method or field loading.

Another change in the bytecode representation results from removing basic blocks from methods. The advantage of this procedure is explained in Section 4. A basic block is a straight-line piece of code without any jumps or jump targets in its interior. A basic block is always entered at the first instruction, and executes all instructions to the end. We have chosen the granularity of basic blocks as the smallest unit of optimistic elimination and on-demand recovery.

As can be seen in Fig. 7, the SlimAnalyzer identifies basic blocks that are related to exception handling. If we remove a basic block from the actual bytecode, some compensating work has to be done: First, all instructions of this basic block are replaced by just one placeholder instruction, as shown in Fig. 8. This instruction references to the corresponding basic block with a numeric identifier. Second, this basic block has to be stored in a separate data structure on the server in order to be able to answer the request of the mobile device if this basic block is requested on demand. Third, all jump instructions of a method have to be updated if a basic block is removed. A jump instruction in Java is followed by a branch offset. The branch target has to be within the same method, but a negative or a positive branch offset is possible. This means that the branch target could be anywhere in the method. The most tricky part is to keep all branch instructions up to date during the linking process since each removed basic block changes the length of a method. We replace each removed basic block with an instruction called `block`, followed by the unique basic block identifier. If such an instruction is reached during execution, the corresponding basic block will be requested from the server and transferred to the mobile device.

If we perform all the optimizations that are mentioned in this section, we change the ratio between meta-information and bytecode from about 80:20 to 30:70 as can be seen in Fig. 6. The overall size of these applications can be found in Section 5.1.

3.1. Client-Side VM modifications

We have based our work on Sun's KVM [19], a Java virtual machine targeted for resource-constrained devices. The 'K' in KVM stands for kilo because it is only a few 100 KB in its standard configuration.

In particular, following changes are required by our SlimVM vs. KVM's data representations:

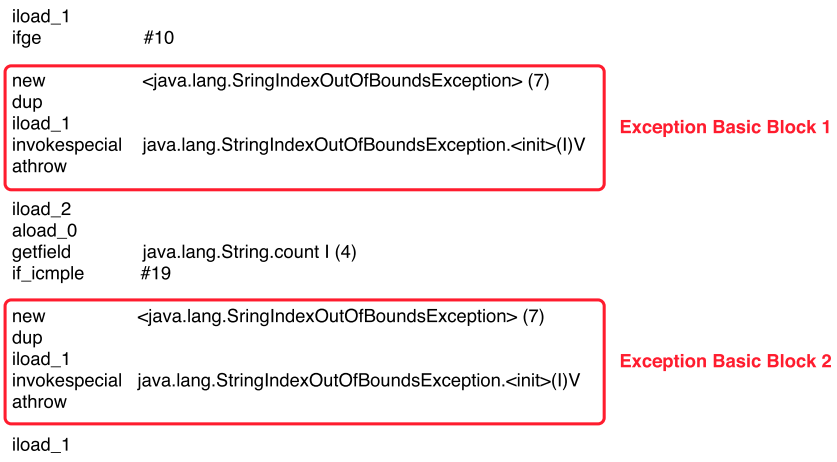

Fig. 7. Instructions and identified exception basic blocks. Each basic block gets a unique identifier.

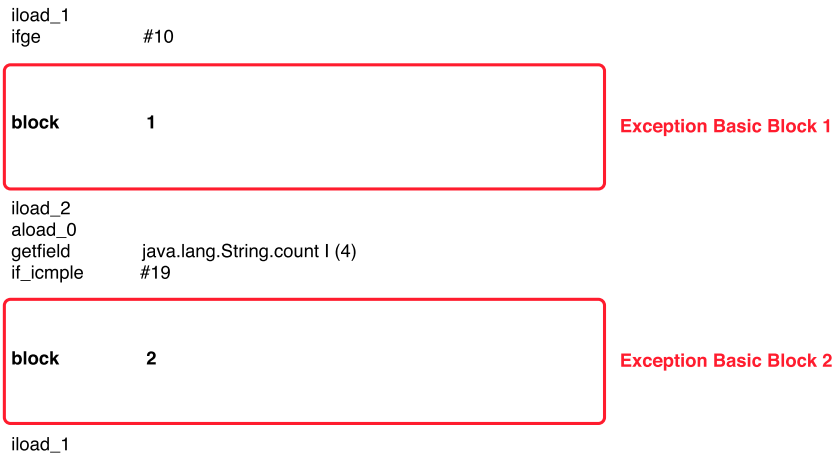

Fig. 8. After bytecode modification, each exception basic block is replaced by an instruction called `block` and the numeric identifier for the exception basic block.

- A common class in KVM is identified by its package and base name. There is no need to keep these items since we now use a numeric class identification. We generate an array that stores the pointer to a class and each class can be requested by a simple array lookup. The address of class five, for example, is stored on position five of the class array. So we do not need a complex class lookup that maps the package name and base name strings into a hash table as KVM does.
- KVM loads all constant pool information into a predefined structure during class loading. SlimVM loads a separate stream that contains all remaining strings and a lookup table for unresolved methods.
- KVM uses a complex method construct with all meta-information that is needed for its constant pool reference technique. SlimVM in contrast uses only a bytecode stream that includes all methods for each class. Information that is needed by the virtual machine is directly inserted into the bytecode stream. Each method starts with its length, frame size, number of arguments, and needed stack size.
- Like the above, the field table is also replaced with a data stream. All references in field instructions are replaced by offsets into this stream during the linking process.
- The whole SlimVM architecture is designed for "friendly" users and code verification is done on the server host. The connection between the server host and the mobile client has to be secure and all code that is executed on the mobile device is secure.

Due to the fact that no complex constant pool or method structures have to be generated during class loading, we get a much simpler class loader.

Further changes result from moving integer and floating-point values from the constant pool directly into the bytecode stream. A single byte that is inserted right after the opcode that identifies the type of the value that is related to this opcode. Long and double values are also loaded using this strategy.

The original class loading technique changes completely. Instead of opening and reading all content of a single class file SlimVM has to send a class-request to the server. The response contains all basic information about the class including static and instance constructor methods.

A new opcode needs to be added for loading basic blocks. If this opcode is reached during execution, a request with the corresponding basic block identifier is sent to the server. The returned bytecode stream is stored separately and the reference from the load instruction is rewritten to this new bytecode stream. Possible branch instructions at the end of the new block need to be redirected to the appropriate instructions. Furthermore, affected exception tables have to be updated.

3.2. Native functions

Another lookup mechanism that has to be adapted considers native functions. KVM does not support the full specification of the JNI. Instead, it uses a limited version that is designed to be less complex and smaller. In the KVM all lookups for native functions are done with the name of the method. Since these names are removed from the constant pool, another lookup technique has to be found.

We changed the lookup mechanism again to numerical identifiers. According to the KVM specification, all native methods have to be present at the virtual machine. The lookup is done with an identifier for the class where the method can be found and an identifier for the method itself. The actual lookup is done in two steps:

- Find the correct class.
- Find the correct function.

All classes that have native functions have to be identified with the same number at each application start. Therefore, these class files are always added to the class handler in the same order and so they always get the same numeric id. Once an appropriate class is found, the lookup continues at the method lookup structure.

The resulting client-side version of the VM is much smaller and less complex. The size of the original KVM is about 250 KB for a standard Unix build. Due to the movement of the verifier, class loader, parts of the linking process and optimizing strategies to the server side, the remaining size of our SlimVM on the client side is about 150 KB.

3.3. Fixed and dynamic cost

The memory footprint of an application is not only affected by its code and library. We also have to consider the cost of the virtual machine itself that executes the application. There are two main categories of memory allocations that we can separate. First, the fixed cost represent the virtual machine itself. This includes all the code and data structures for the internal parts such as class loader, interpreter, garbage collector, verification process, initialization process etc. Dynamic costs on the other hand represent the footprint that is related to the application including all classes, bytecode, object information, objects, exception tables, etc.

Fixed costs are more or less relevant depending on the size of the application. For a simple "HelloWorld" application, the fixed costs dwarf the size of the application. For other benchmarks such as JavaGrande, the fixed costs are about 1% compared to the allocations of the application itself. The original approach of KVM and most other virtual machines requires to load and initialize classes at start up even if they are not needed for the execution at all. We also load the same classes but since we do not separate classes that are loaded during the initialization of the VM and classes that are loaded at run time, we only load the basic information for the classed needed during the initialization process. Methods are only shipped with basic information if we see them referenced from the application. This saves transfer size for the startup of SlimVM which also implies a faster VM initialization process and smaller fixed costs for the VM in general. Another optimization that reduces the transfer size and is not implemented yet would be to store all basic information that has to be loaded at each startup on the client instead of transferring it from the server. The server would decide if the pre-stored information should be loaded or a new version from the server has to be requested.

4. Partial and demand-driven loading

Current Java environments mandate that a file has to be fully transferred before it can start executing. This is also known as strict execution semantics as part of the run-time system. Some work has been done to overcome this problem [11]. For large files and low bandwidth, the delay could be significant. In our approach, the class file is fully loaded in our server environment. When a class file is requested, only the basic information like static and instance variable size, static and instance constructor and remaining strings in the constant pool are transferred. This allows us to continue execution right after this small block of data is transferred.

Effective partial loading and on-demand code addition has the goal of transferring only data and meta-information that is absolutely needed during execution. The idea is to leave data that is less likely to be needed on the remote host and request it on demand.

Our analysis considers three different levels of granularity: a Java class, a single method, and a single basic block.

4.1. Class-level analysis and transfer

If the atomic unit of code transfer is the class, then we have a situation similar to the standard Java mechanism, except that our class files are smaller due to our reduction of the constant pool.

Our class-level analysis collects all unmodified standard Java class files required by the KVM. As mentioned in Section 3, we have to deal with a subset of 73 class files that are possibly needed at run time to execute the simple `HelloWorld` application. This set is generated by collecting all constant pool class references. The actual value may be higher if there are additional classes loaded by native code.

If we log all class files that are loaded, we get the following set:

- package java.lang: Object, Class, String, System, Thread, Throwable, Error, OutOfMemoryError, StringBuffer, Character, Runnable, VirtualMachineError
- package java.io: OutputStream, Writer, PrintStream, OutputStreamWriter
- package com.sun.cldc.io: ConsoleOutputStream
- package com.sun.cldc.io.i18n: Helper, StreamWriter
- package com.sun.cldc.io.i18n.uclc: DefaultCaseConverter
- package com.sun.cldc.io.i18n.j2me: ISO8859_1_Writer
- application package: HelloWorld

These are 22 class files for a normal execution without any exceptions or other errors. `Object`, `Class`, `String`, `Thread`, `Throwable` and `Error` are loaded during initialization of SlimVM. As mentioned before, low coupling between class files and methods leads to much better optimization potential. This is just one reason why we use Sun's KVM because they have constructed a completely new library with a focus on a low degree of internal coupling.

We use a simple approach to predict which classes are really needed. If a reference to a specific class is needed and this class has not yet been transferred to the mobile device, we send a request to the server and fetch it. In the majority of cases, this happens when the opcode "new" is executed. There are also some cases in which an array of objects is initialized or static variables or methods are called. This approach assures that we transfer only the minimum set of classes to the embedded system.

In order to get comparable results, we perform the same approach with our new SlimVM data format. The amount of data that is actually transferred to the mobile device is reduced dramatically due to the elimination of the constant pool and most meta-information and the pre-linking of bytecode. Results are discussed in Section 5.

4.2. Method-level analysis and transfer

Alternatively, one can focus on methods as the unit of analysis and transfer. This is a very effective strategy when only small portions of large class libraries are used. Our approach is that a class contains all the information about its superclass, access rights, strings that are left in the constant pool, fields and initial methods such as `init` and `clinit`. If a class is loaded, these items are transferred to the mobile device. Depending on the execution mode (optimistic or pessimistic approach), other methods might be part of the basic class information.

As mentioned before, we have implemented two strategies:

- **Pessimistic approach:** Every method that is referenced by an `invoke` statement is shipped to the mobile device during basic class loading. We obtain this subset of methods by traversing all `invoke` instructions and removing never invoked methods. This has to be performed several times since removing one method might cause another method to be unreferenced.
- **Optimistic approach:** Each class is transferred with its initialization methods `init` or `clinit`. All other methods have to be requested at run time. This is very effective if there is a class requested because of a static field.

4.3. Basic-block-level analysis and transfer

Our final analysis focuses on basic blocks as the unit of code transfer. Our analysis tries to identify those parts of individual methods that are surely going to be executed. Blocks of if–then–else statements, exception parts or dead code that can never be reached might be a target for elimination under this approach. A very aggressive dead code detector can be used since we are able to dynamically provide basic blocks on demand later.

To estimate possible savings by removing exception basic blocks, we analyzed how much exception code is contained in our benchmark applications. We obtained the following results as shown in Table 1.

Our results indicate that about 10%–13% of the whole application and library bytecode is exception code. It might be even higher since we do not consider functions that are only called from exceptions. Since exception code is less likely to be executed, these parts remain on the network host and are supplied only on demand.

5. Results

One of the initial claims of this paper was to show that it is possible to predict with some confidence which parts of a library are needed at run time. We discuss this claim in Section 5.1. There are also performance results that we discuss in Section 5.2. Finally, we show the impact of network delays in Section 5.3.

Table 1
Exception code in benchmark applications. By dynamically providing exception basic blocks on demand, we can realize a code size reduction of about 10% if all library code is needed during execution.

HelloWorld	13, 45%
Linpack	12, 49%
Compress	12, 18%
Scimark2	11, 40%
JavaGrande	11, 70%
Sievebits	13, 33%
Matrix	13, 32%

We have completed a prototype implementation of the SlimVM system, including the SlimAnalyzer tool for the network host. All our space related benchmarks were run on a Dell Inspiron 1501 using JDK 1.4.2 for Linux. The SlimAnalyzer tool is based on BCEL version 5.2.

Our benchmark section is split into two parts. In order to show code size savings, we run all benchmarks on the same laptop and simulated network transfers with file accesses. In order to measure the impact of network delays, all benchmarks of section Section 5.3 were run in a distributed environment that consists of a Acer TravelMate 4502 as the client and an Intel 2.33 GHz Dual Core Processor (5140) system running Red Hat Enterprise Linux 4 and Linux kernel 2.6.9-55.0.6.ELsmp as the server. SlimVM (client) and SlimAnalyzer (server) are connected via C Sockets over a wireless network environment.

In order to be able to run benchmarks with the library of KVM and make consistent comparisons, we adapted some of the SpecJVM benchmarks.

All generated subsets of applications were tested in order to ensure that they execute correctly in their reduced form. The SlimAnalyzer tool only needs to get the name of the class in which the main method is located and SlimVM has the IP address of the server as input.

The SlimAnalyzer tool has four different execution modes:

- Pessimistic approach with method granularity.
- Pessimistic approach with basic block granularity and removal of all exception basic blocks.
- Optimistic approach with method granularity.
- Optimistic approach with basic block granularity and removal of all exception basic blocks.

5.1. Space savings

To get an overview about the individual contributions of the optimizations discussed in this paper, we start with Fig. 9 that shows size reductions for each benchmark application when going from the class-level granularity down to basic block granularity. KVM has a library with 177 class files. A simple `HelloWorld` application consists of 178 class files. Starting with the class-level approach, we get a reduction of about 50% by loading only the needed classes. Under the heading "class files", our figure shows the ratio between all library and application sizes vs. class files that are loaded at run time. These numbers are based on the standard Java class file format. All further optimization results are based on the SlimVM data transfer format.

The next optimization deals with reducing the constant pool. As described in Section 2, we perform pre-linking whenever possible. References that cannot be resolved statically are stored in a compressed lookup table. In addition to the class optimization in which only initialized classes are loaded, we get a reduction of up to 92%.

Our class file and constant pool optimizations use the same class granularity approach. They only vary in the transfer format that is used between the server and the client. We get a remarkable reduction of transfer size by just changing the transfer format.

The next step is to minimize the bytecode. As mentioned in Section 3, we change the atomic unit of loading from classes to methods. Fig. 9 shows that this saves another 3% for a `HelloWorld` application. We now obtain a reduction up to 95%. `Scimark2`, which uses more library functionality, shows a reduction of 84%.

We claimed in Table 1 that between 10% and 13% of the application bytecode is related to exception basic blocks. If we just consider methods that are actually loaded at run time, we get results as shown in Table 2. Our last optimization is based on a basic block granularity and each exception basic block is removed and has to be requested on demand. On the right-hand side of Fig. 9 there are the results for this optimization. We save between 1% and 3% for each benchmark.

Tables 3–7 show detailed information about data that is transferred for each individual benchmark. The first row shows the size of the subset of classes that are collected during static analysis if we follow all references. The second row shows the results for our first execution option. If we take a closer look at Table 3, we consider 8359 bytes as most likely needed and miss 535 bytes for the pessimistic approach. This means we have to reload additional information for several classes. If we sum up the loaded and reloaded bytes we get the total bytes that are transferred for this benchmark. The last column shows the relative savings to the original subset of classes that provide a full coverage of all needed functionality. The following rows show the results for the other execution modes. The more aggressive we perform cold code removal, the more we reduce the transfer size but with the penalty of more reloads. All benchmarks show an increase of reloads for more optimistic cold

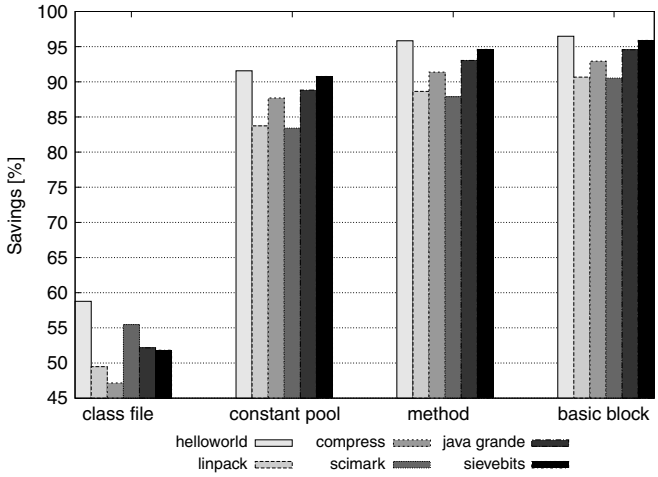

Fig. 9. (Note the modified origin.) Space reduction achieved by performing a series of optimizations. Class file: transferring just the class files that are referenced in the constant pool. Constant pool: load only class files needed during execution. All constant pool references are removed and replaced by a small lookup table. Methods: analyze method invocations and remove all methods that are never invoked. Basic blocks: build a basic block graph and remove all exception basic blocks.

Table 2
Exception code in benchmark applications considering just loaded methods. Here we only consider actual loaded methods at run time instead of all library functions as shown in Table 1. By dynamically providing exception basic blocks on demand, we reduce the bytecode by up to 17%.

HelloWorld	17, 52%
Linpack	8, 14%
Compress	10, 33%
Scimark2	7, 13%
JavaGrande	9, 81%
Sievebits	16, 33%
Matrix	15, 22%

Table 3
SlimVM subset space savings for HelloWorld. (XBB stands for Exception Basic Blocks).

Library subset (KVM)	Loaded	Reloaded	Whole	Improvement
Original (KVM)	125215	0	125215	0
Pessimistic with XBB	8359	535	8894	92, 90%
Pessimistic prune XBB	7435	447	7882	93, 71%
Optimistic with XBB	4827	1855	6682	94, 66%
Optimistic prune XBB	4774	1396	6170	95, 07%

Table 4
SlimVM subset space savings for Linpack.

Library subset (KVM)	Loaded	Reloaded	Whole	Improvement
Original (KVM)	130069	0	130069	0
Pessimistic with XBB	22751	624	23375	82, 03%
Pessimistic prune XBB	21356	536	21892	83, 17%
Optimistic with XBB	9592	9223	18185	85, 53%
Optimistic prune XBB	9539	8352	17891	86, 24%

code removal. If we look at the optimistic execution mode without the exception basic blocks, we get a reload rate of about 22% for the `HelloWorld` application. This goes up to about 46% for the `Linpack` benchmark as can be seen in Table 4. This can be explained with the higher use of library code for these benchmarks. On the other hand, since only 46% of the library code is used for classes that are actually loaded, we can see that there exists a high potential of code transfer on demand. As we see from the comparison of the HelloWorld and Linpack example, the pessimistic approach does a better job if more library functionality is used by the application.

Table 5
SlimVM subset space savings for 201 compress.

Library subset (KVM)	Loaded	Reloaded	Whole	Improvement
Original (KVM)	139419	0	139419	0
Pessimistic with XBB	15507	1798	17305	87, 59%
Pessimistic prune XBB	14476	1583	16059	88, 48%
Optimistic with XBB	8662	6238	14900	89, 31%
Optimistic prune XBB	8609	5561	14170	89, 84%

Table 6
SlimVM subset space savings for scimark2.

Library subset (KVM)	Loaded	Reloaded	Whole	Improvement
Original (KVM)	140079	0	140079	0
Pessimistic with XBB	25411	478	25889	81, 52%
Pessimistic prune XBB	23986	449	24435	82, 56%
Optimistic with XBB	11579	8527	20106	85, 65%
Optimistic prune XBB	11526	7736	19262	86, 25%

Table 7
SlimVM subset space savings for java grande single 2.

Library subset (KVM)	Loaded	Reloaded	Whole	Improvement
Original (KVM)	140543	0	140543	0
Pessimistic with XBB	14625	504	15129	89, 24%
Pessimistic prune XBB	13685	416	14101	89, 97%
Optimistic with XBB	6742	4846	11588	91, 75%
Optimistic prune XBB	6681	4379	11060	92, 13%

Table 8
Bytecode transferred for benchmark applications. The first column represents the standard Java bytecode size that is needed if all possible needed class files are loaded. All other columns represent bytecode size that is transferred for our approaches in the SlimVM data format. XBB stands for exception basic blocks.

	Whole	Pessimistic	Pessimistic (prune XBB)	Optimistic	Optimistic (prune XBB)
HelloWorld	26700	5777	4765	3565	3053
Linpack	28883	18225	16742	13665	12741
Compress	29796	12064	10818	9659	8929
Scimark2	31927	20403	18949	14620	13776
JavaGrande	31218	10477	9449	6936	6408
Sievebits	27062	7421	6209	4586	4060
Matrix	26964	6649	5637	4350	3838

Table 8 shows the bytecode size for each optimization in absolute numbers. We start with the whole standard Java bytecode size of all referenced class files in the original constant pool. Then we change to our SlimVM data format. All bytecode that is transferred in each execution mode is presented in the columns.

Fig. 10 shows the achieved savings for the method granularity. More complex applications such as `scimark2` need more library functionality and therefore more classes are loaded. As can be seen in the results, some never invoked methods are transferred. Huge classes with many methods cause reduced savings. A better job is done by the optimistic approach. Each method is loaded on demand but this can cause some delays during execution as can be seen in Section 5.3.

Fig. 11 shows the achieved savings for basic block granularity. We obtain better results for each of the benchmarks. In comparison to Fig. 10 we gain between 1% and 3% by replacing exception related basic blocks with a single instruction that loads this code section if needed.

Finally, Fig. 12 presents absolute numbers of transferred bytes for each approach. A simple `HelloWorld` application needs about 7K of memory in the smallest approach. KVM needs 125K if all class files that are possibly needed are transferred or about 88K if all class files that are loaded at run time are transferred. Without analysis, one would need to transfer the whole library of about 187K.

Another very interesting outcome of our approach is the overall memory consumption during execution. If we monitor all memory allocations for the startup of the virtual machine and the actual program execution, KVM requests 99K for a simple `HelloWorld` application. SlimVM in contrast requests only 24K during the smallest execution mode (optimistic approach with basic block granularity and removed exception basic blocks). This is a reduction of 75%.

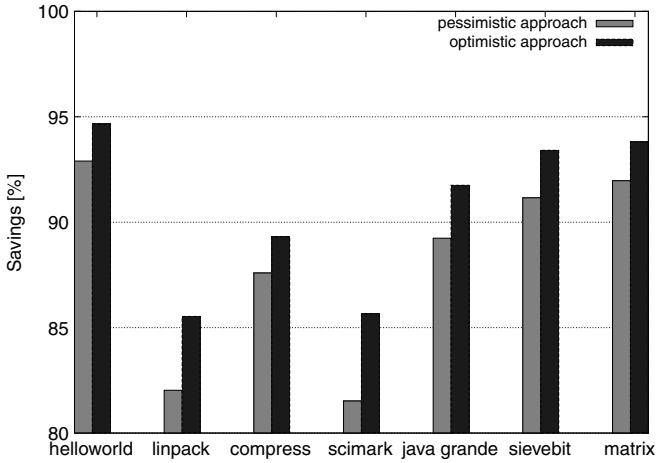

Fig. 10. Actual savings for method granularity. We save up to 94% of bytes transferred in comparison to all application and library files that have to be transferred if a standard JVM is used.

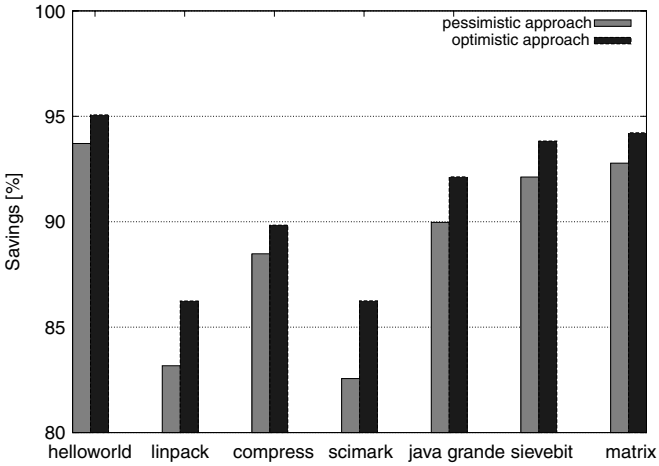

Fig. 11. Actual savings for basic block granularity. We save up to 95% of bytes transferred in comparison to all application and library files that have to be transferred if a standard JVM is used.

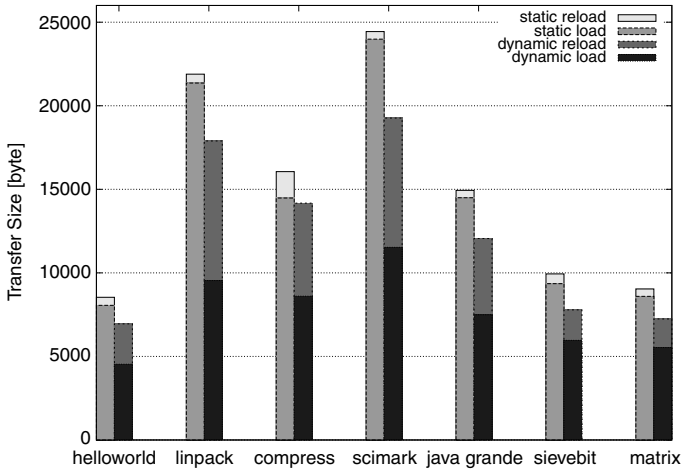

Fig. 12. Transfer size for optimistic and pessimistic heuristics. Load represents basic class information and initialization methods. Reloads referred to the dynamic provision of missing methods and basic blocks during execution. Static load and static reload is a pessimistic approach where never invoked methods are removed, but there are still some methods which must be transferred since they cannot be predicted statically. Dynamic load and reload is an optimistic approach where only initialization methods are transferred at the beginning. All other methods have to be requested by the mobile device on demand. Communication overhead is included in our figures.

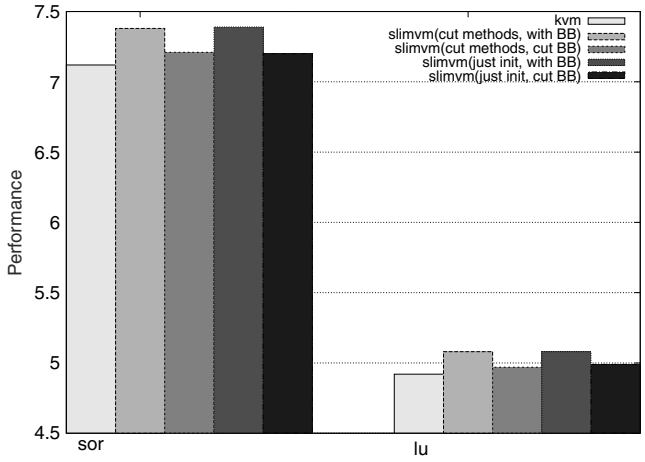

Fig. 13. Performance of KVM vs. SlimVM. The numbers represent the performance output of the `sor` and `lu` benchmark. A slight performance increase can be measured. Some of this is due to off-target verification, reduced constant pool lookups, and off-target linking. These benchmarks do not include network transfers.

5.2. Effect on performance

Our approach is not focused on increasing space efficiency only. Instead, for usability reasons, the trade-off between speed and size should be balanced. In this section we discuss the impact of our modified bytecode representation. We compare all execution modes with the unmodified version of KVM.

SlimVM and SlimAnalyzer run on a single laptop and network requests are changed to file accesses. Every benchmark shows a slight performance increase. Fig. 13 shows the outputs of the benchmarks `sor` and `lu` from the standard `scimark2` benchmark. Analysis and initial transfer time are not included. As expected, off-target linking, removal of constant pool lookups and off-target verification slightly increases the performance. The original KVM rewrites method invocation with so called "fast bytecodes" once the constant pool lookup was performed and the location of the method was found. Whenever possible, we move this time-consuming lookup to the server side and already transfer a rewritten bytecode.

The following reasons are responsible for a slightly better performance:

1. Simpler class loader. SlimVM does not load whole class files.
2. No time-consuming constant pool generation.
3. Almost no method lookups at run time.
4. Off-target verification.

5.3. Latency impact

The last part of our benchmark section deals with impacts of network latencies caused by code reloads. Our approach only makes sense if the overhead for startup and reloads is in proper relation to the achieved memory savings.

The original KVM needs about 10 ms to execute a `HelloWorld` application. A simple `HelloWorld` execution on our distributed SlimVM environment needs about 500 ms. This includes the startup and initialization of the virtual machine, loading library and application code from the server and execution of the code. The execution time also depends on the quality of the network. All benchmarks were run one hundred times and a mean value was taken as the normal execution time.

Table 9 shows the number of reloads during the execution of all benchmarks. Even for the pessimistic approach where we try to catch all possible method calls during the analyzing process, we miss some methods in classes that are loaded with native code. In a simple `HelloWorld` these are 8 missing method calls. For the optimistic approach we have to reload 29 methods during execution for the same `HelloWorld` application.

Fig. 14 shows the resulting trade-off between time and space for the `HelloWorld` application. When SlimVM connects to the server, a time log starts at the server side. Each class or method request is logged using the `rdtsc` x86 assembly instruction. This instruction reads the time step counter that represents count of ticks from processor reset. We need such an exact timing mechanism since the differences between each execution mode are within some milliseconds. As for the `HelloWorld` application, the total execution time is about 0.5 s. Both execution modes (with and without exception basic blocks) for the pessimistic approach are almost equally fast but as can be seen in Fig. 14, SlimVM requests about one kilobyte less data. The optimistic approach transfers about 2–3 KB less, but since we have to perform more server requests, the execution time increases.

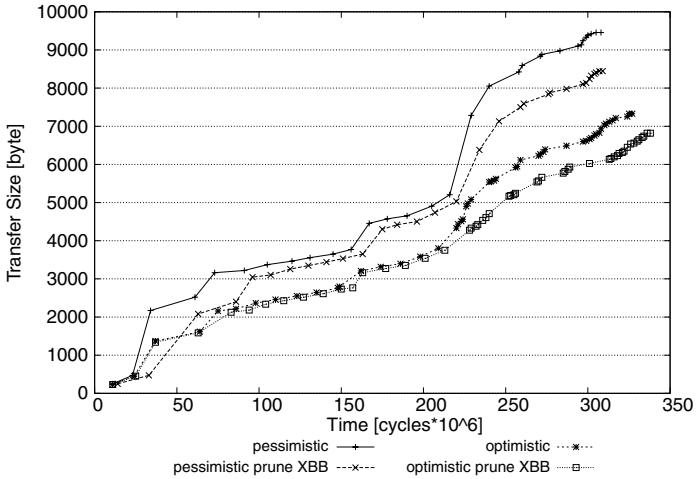

Fig. 14. Trade-off between space and time for a simple `HelloWorld` application. Different execution modes result in better performance or less transfer size. Each dot indicates a load or reload event. Our pessimistic approach with less reload events is slightly faster than our optimistic approach. However, the optimistic approach shows a remarkable transferred data reduction without big time overhead. XBB stands for exception basic block.

Table 9
Actual number of reload events for the optimistic and pessimistic approach during execution. Even with marking all possible function calls during the analyzing process with the pessimistic approach, we miss some functions that are invoked natively in the JVM. For a `HelloWorld` application for example, we have to request 29 functions during execution when using our optimistic approach.

	Pessimistic	Optimistic
HelloWorld	8	29
Linpack	10	62
Compress	16	75
Scimark2	9	78
JavaGrande	7	64
Sievebits	9	56
Matrix	9	49

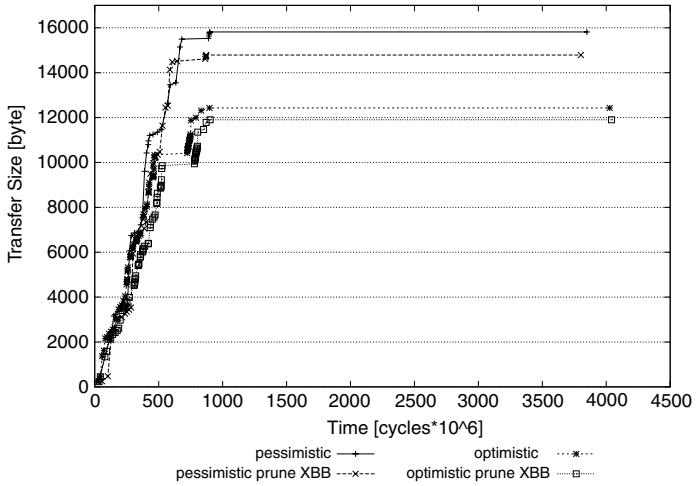

Fig. 15. Trade-off between space and time for the `JavaGrande` benchmark. The very last dot for each benchmark indicates the end of the execution where no actual data transfer was performed. Again, the pessimistic approach is faster than the optimistic approach with the penalty of a bigger transfer size.

Finally, Fig. 15 shows the trade-off between time and space for the `JavaGrande` [12] benchmark. This benchmark is much more CPU orientated. There are only load and reload events at the beginning of the application. No class or method requests are sent to the server after the startup phase. Again, the pessimistic approach is faster since most of the required methods are transferred during class loading. If we transfer each function on its own, as the optimistic approach does, SlimVM has to stop execution and request a method 64 times as can be seen in row 5 in Table 9 instead of the 7 times for the pessimistic approach.

6. Related work

Our work touches on several areas of related work, from alternative class file formats and execution models on the one hand to methods for predicting which parts of an application and libraries are needed during the application's execution on the other hand.

Most of the Java research is focused on making execution faster rather than making code fit into a smaller space. However, the size of Java class files has long been recognized as an issue for embedded devices. The first and maybe most simple idea to overcome this problem is to compress the class files. One well known solution is the *jar* file format. Jar files are collections of individually compressed class files. They are about half the size of the original class files.

Pugh [13] describes a wire format that reduces these jar files by another 20% to 50% on average, using the gzip algorithm. The wire format reduces redundant information in the constant pool by merging the individual class files. As a result, this approach cannot deal with partial class loading very well. Java Specification Request 200 (JSR-200, "Network Transfer Format for Java Archives") [10] is another dense download format co-invented by Pugh. It was proposed to reduce the download size of the JRE installer for windows. It takes all the class file data, stores them into a single file and reorganize them. Pack200 assumes that the file is post-processed with a compression tool like gzip.

Another example for an alternative file transfer format is the J9 Java Executable (JXE) format [23]. It is specifically designed to load Java applications on embedded devices. Similar to our work it also performs constant pool compression, off-target linking and other space optimization techniques. Bytecode is compiled to a target specific native code before the program is run and stored in ROM. The virtual machine can use memory mapping in order to load the executable.

Slim Binaries [8] use predictive compression on abstract syntax trees. This approach is based on the fact that different parts of a program are often similar to each other. Using this approach, the size of a complete application can be reduced by a factor of 3. Stork [17] uses abstract syntax trees for a new mobile code format. He shows that the size of regular Java archives can be reduced by a factor of 3 to 8, resulting in improvements of 5%–50% over Pugh's compression.

Bradley [2] is the author of the jazz file format that far exceeds the compression rate possible for JAR files. He collects Java class files, uses different data compression methods and achieves compression of up to 70%.

Ernst et al. [7] describe an executable representation for bytecode called BRISC that can be interpreted *without* decompression. It has the same size as gzipped x86 programs.

Clausen et al. [5] factorize common instruction sequences and achieve a reduction of 85% of the original bytecode size on average. However, since about 80% of a class file is meta-information, only a 20% reduction of the whole class file is possible. In different work [4], the authors show that side-effect analysis with dead code elimination and loop-invariant code motion results in a speed increase of up to 25% in execution time.

Yang [25] 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 [3] has shown that approximately 3% to 16% dead or unreachable code exists in programs. Tip and Sweeney [22,21,23] reduce the size of Java applications by using an application extractor. Like our approach, their aim is to ship only needed class files to a mobile device, cut out never invoked methods, and perform some name compression of the constant pool. Since they do not support dynamic addition of code, they have to use a pessimistic approach, which results in a reduction of only 37% of the original size by cutting unused library functionality. In contrast, we get much better results by using optimistic analysis and dynamically providing code when needed.

Krintz et al. [11] developed a system in which execution overlaps with transfer. While there is no focus on reducing transfer size, the average reduction in overall execution time is up to 40%.

Rayside et al. [15] explain a number of techniques for reducing the size of Java class files. They show that the most significant size reduction can be achieved by reducing the constant pool. Typical results are 25%, and about twice that (50%) when using a single constant pool for all class files in a jar archive.

To understand the impact of required library code and the importance of analysis for code actually required by an application, it is instructive to look at Rayside [14], where the authors explain how to reduce a HelloWorld application from 8693 KB (combined library and application code) to 535 KB by analyzing only used class files and removing never invoked methods.

Titzer et al. [24] present a dynamic VM named ExoVM where all code, data and VM features are packed into a binary image. 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 the non-heap memory allocation can be reduced by up to 75%.

More recent work by Sartor et al. [16] compares different heap data compression techniques. It shows that using optimizations like deep object equality, field value equality, removing bytes that are zero, and compressing fields and arrays with a limited number and range of values can reduce the memory footprint by up to 86% and on average 58%.

7. Conclusions and outlook

We have presented a framework to reduce the memory consumption of Java applications that are deployed on connected mobile devices. By performing off-target linking we are able to eliminate the need to store meta-information such as textual class or method names on the target device.

Our prototype system improves the state of the art in Java code compaction in two important ways. First, we perform optimistic compaction and only download code to the device that has a high probability of actually being needed at run time. Second, since Java is a highly dynamic language with reflective capabilities, this can lead to some small remaining parts of the program code that are then missing in the target system when execution starts. We use on-demand partial loading to satisfy such additional dependencies at run time.

Our system also significantly differs from previous work in that we use basic blocks as the smallest deployment unit. This allows us to not only eliminate unneeded classes or methods, but even unneeded (or infrequently needed) method parts such as exception handlers. Our off-target linking and code compaction approach also reduces the overall memory utilization of the virtual machine (in some cases by as much as 75%). In addition, we also observed a slight performance gain in a locally simulated environment. Our approach may even be useful in some non-distributed settings but this issue will require further study.

As far as future work is concerned, we are particularly 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 that 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.

Acknowledgements

This research effort is partially funded by the National Science Foundation (NSF) under grants CNS-0615443 and CNS-0627747. The US 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 US Government.

References

[1] D.N. Antonioli, M. Pilz, Analysis of the Java Class File Format. ifi-98.04, Department of Computer Science, University of Zurich, Apr. 28, 1998.
[2] Q. Bradley, R.N. Horspool, J. 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, IBM Press, 1998, page 7.
[3] J.A. Butts, 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, ACM Press, New York, NY, USA, 2002, pp. 199–210.
[4] L.R. Clausen, A java bytecode optimizer using side-effect analysis, Concurrency: Pract. Exp. 9 (11) (1997) 1031–1045.
[5] L.R. Clausen, U.P. Schultz, C. Consel, G. Muller, Java bytecode compression for low-end embedded systems, ACM Trans. Program. Lang. Syst. 22 (3) (2000) 471–489.
[6] M. Dahm, Byte Code Egineering Library (BCEL), 2001. http://jakarta.apache.org/bcel/.
[7] J. Ernst, W. Evans, C.W. Fraser, T.A. Proebsting, S. Lucco, Code compression, in: PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1997, pp. 358–365.
[8] M. Franz, T. Kistler, Slim binaries, Communications of the ACM 40 (12) (1997) 87–94. Also published as Technical Report TR 96–24, Department of Information and Computer Science, University of California, Irvine, June 1996.
[9] J. Gosling, B. Joy, G. Stelle, The Java Language Specification, in: The Java Series, Addison-Wesley, Reading, MA, 1996.
[10] W.P. John Rose, Kumar Srinivasan, JSR 200: Network Transfer Format for JavaTM Archives, September 2004.
[11] C. Krintz, B. Calder, H.B. Lee, B.G. Zorn, Overlapping execution with transfer using non-strict execution for mobile programs, in: ASPLOS-VIII: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, 1998, pp. 159–169.
[12] J.A. Mathew, P.D. Coddington, K.A. Hawick, Analysis and development of java grande benchmarks, in: Proc. of the ACM 1999 Java Grande Conference, San Francisco, 1999.
[13] W. Pugh, Compressing java class files, in: PLDI'99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1999, pp. 247–258.
[14] D. Rayside, K. Kontogiannis, Extracting java library subsets for deployment on embedded systems. csmr, 00:102, 1999.
[15] D. Rayside, E. Mamas, E. Hons, Compact java binaries for embedded systems, in: CASCON'99: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 1999, page 9.
[16] J.B. Sartor, M. Hirzel, K.S. McKinley, No bit left behind: the limits of heap data compression, in: ISMM'08: Proceedings of the 7th International Symposium on Memory management, ACM, New York, NY, USA, 2008, pp. 111–120.
[17] C.H. Stork, Well: A language-agnostic foundation for compact and inherently safe mobile code. Ph.D. Thesis, University of California, Irvine, School of Information and Computer Science, Irvine, CA, USA, 2006. Adviser-Michael Franz.
[18] SUN J2ME's Homepage. http://java.sun.com/javame/.
[19] Sun Microsystems. KVM - Kilobyte Virtual Machine White Paper. Palo Alto, CA, USA, 1999. http://java.sun.com/products/cldc/wp/.
[20] Sun Microsystems Inc. Connected, Limited Device Configuration, and the K Virtual Machine, April 2000.
[21] P.F. Sweeney, F. Tip, Extracting library-based object-oriented applications, in: SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, New York, NY, USA, 2000, pp. 98–107.
[22] F. Tip, P.F. Sweeney, C. Laffra, Extracting library-based java applications, Commun. ACM 46 (8) (2003) 35–40.
[23] F. Tip, P.F. Sweeney, C. Laffra, A. Eisma, D. Streeter, Practical extraction techniques for java, ACM Trans. Program. Lang. Syst. 24 (6) (2002) 625–666.
[24] B.L. Titzer, J. Auerbach, D.F. Bacon, J. Palsberg, The exovm system for automatic VM and application reduction, in: PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2007, pp. 352–362.
[25] Y.-S. Yang, M. sik Jin, S.-I. Jun, M.-S. Jung, A study on an efficient pre-resolution method for embedded java system, in: Virtual Environments, Human-Computer Interfaces and Measurement Systems, July 2004, pp. 20–24.