Laboratoire d'Architecture des Processeurs - EPFL
Semester Project - Transparent PLD use from Java
Final Report

Work:	Christophe Dubach & Alok Menghrajani
Report:	Alok Menghrajani
Assistant:	Miljan Vuletic
Professor:	Paolo Ienne

February 2004

Special thanks to the former students Cédric Gaudin, Stéphane Magnenat, Julien Pilet and Damien Baumann for their continuous support. To Miljan Vuletic and all the LAP professors without whom nothing would have happened.

Contents

1 About This Document

I am a student at EPFL (Swiss Institute of Technology at Lausanne), Switzerland. I am doing a Master in Computer Science. This document is the report of my final year's semester project: "Transparent PLD use from Java". I worked at the Laboratoire d'Architecture des Processors.

Christophe Dubach was assigned a similar project ("Java virtual machine on FPGA-based flatforms"), we therefore collaborated together and this is the result of our combined work.

The CD-ROM included with this report contains the source code of all our work. It also includes all the tools and libraries needed for developement. Installation instructions are included in our report entitled: "RokEPXA Development Environment From Scratch" which is also being submitted simultaneously.

All the tools and libraries we used are available as part of the various open source projects, which are constantly evolving and being updated. We recommend using exactly the versions which we have used. We had to correct (patch) some of these tools. Newer version might not include these corrections.

All our work is being released under the GPL license, which means you can copy, read, use, modify our work, but you MUST keep it under the GPL license (which means you must also share any work based on our's). Christophe and I both strongly believe in open source development, we hope our contribution to the community will be helpful.

Part of my work is based on software developed by the Apache Software Foundation (http://www.apache.org/).

2 Introduction

2.1 Embedded Systems

The trend in the current society is to have smart chips in everyday objects. These chips are called embedded systems because they control one or more functions of the equipement. They are typically required to meet very different requirements than a general-purpose personal computer (they usually have small amount of memory, are cheap to produce and must consume low power).

These chips can be categorized in two types:

1. Processors (which can perform general tasks)

2. Specialized chips designed for specific tasks

Designing specialized chips is complex and expensive, but it is often the only way to meet real-time constraints (eg: a dsp[1] chip or 3d graphics card can outperform a fast processor because it can be designed to perform multiple calculations simultaenously).

For research and development applications, there is a special kind of chip called FPGA[2] which can be programmed at the circuit level to perform a specific task. The programming is done in the same way as for designing a chip; that's is in a hardware description language (eg: VHDL[3]). The flexibility offered by FPGA chips comes at the cost of extra programming effort.

2.2 RokEPXA

http://lapwww.epfl.ch/dev/arm/

We used the RokEPXA board designed at the Laboratoire d'Architecture des Processeurs, EPFL. This board uses Altera's EPXA *(http://www.altera.com/products/devices/arm/arm-index.html)* integrated circuit which is a hybrid architecture. It combines a processor (ARM922T) with a FPGA.

The typical usage scenario for the RokEPXA is for robots; the FPGA controls the fundamental functions (sending signals to the motors, receiving signals from sensors, etc.) and the processor handles the higher level behaviour (articificial intelligence, navigation, etc.).

The FPGA can also be used to connect a camera or ethernet interface.

The FPGA on the RokEPXA can be programmed by the processor (by loading a `.map`[4] design file that is obtained by compiling the VHDL code). This means that the processor and FPGA can share "objects". The processor can perform general computing while the FPGA is programmed to compute time critical sections (coprocessor).

One of the difficulties is communication between the processor and FPGA chip. Intensive work on this has already been done by Miljan Vuletic. He has designed a solution called VMW[5] (formely known as VIM), that allows the coprocessor to access the virtual memory of a a user-space application.

2.3 Past And Present Work

The LAP has been working on the RokEPXA for a couple of years. When we began our project, we had a working RokEPXA card, with a minimal development environment (that we abandoned). We had the design files for IDEA (we'll get back to this later) and network support.

VMW was also in it's completion stage.

[1]digital signal processing
[2]Field-Programmable Gate Array
[3]VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language
[4]In our specific environment, the `.map` files end with the .sbi extension
[5]Virtual Memory Window

We used a lot of open source projects such as the Linux Kernel, the Arm Linux project, etc. (we will introduce these projects and others later in this document). Some of these projects laked testing on our specific hardware and lead us to spend a lot of time debugging.

3 Project Description & Objectives

We set ourselves the following objectives:

1. **Creating a complete developement environement for the RokEPXA.**
   Christophe Dubach & Alok Menghrajani
   Altough there already was a develpement environement, we ran into trouble because it was incomplete, lacked documentation and had various bugs. We created a fresh environement that includes: linux 2.6.0, nfs support, a cross-compiler[6], a native compiler, busybox system tools, a JVM[7], dynamically linked library support and floating-point emulation.

2. **Porting IDEA[8] to Java**
   Christophe Dubach & Alok Menghrajani
   IDEA is an encryption algorithm. We already had a C and VHDL version, so we ported it to Java in order to test our work and mesure performances.

3. **Using the coprocessor from within Java.**
   Christophe Dubach
   Given a .map *file, we want to run it from within Java. This way we can write most of the code in Java and only the time critical part in VHDL. Christophe worked on this, his solution is written in C and called from Java using JNI[9].*

4. **Transparent PLD[10] use from Java.**
   Alok Menghrajani
   Given a Java method and a `.map` file that performs exactly the same calculation, we want the JVM to run the `.map` file instead of the Java version of the code. A PLD is the same as a FPGA.

   "transparent" means that we don't want to have the user modify the Java source code. We ignored the aspect of how to generate the `.map` file from the Java (we will talk about this in the conclusion of this report).

[6] A compiler that runs on our desktop environement but produces code for the ARM
[7] Java Virtual Machine
[8] International Data Encryption Algorithm
[9] Java Native Interface
[10] Programmable Logic Device

4 Developement Environment

I will keep the discussion about the developement environement to the bare minimum. You can refer to our other report "RokEPXA Development Environment From Scratch", submitted simultaneously.

Here is the list of software we used with a little explanation. We unfortunately didn't find any complete distribution that met our needs, we therefore compiled everything on our own.

4.1 Linux Kernel 2.6.0

http://www.kernel.org/

We decided to switch to Linux 2.6.0 (the former development environment was based on Linux Kernel 2.4.19) for stability reasons. We applied the rmk1 ARM Linux patch (http://www.arm.linux.org.uk/) and then another patch called rokepxa (based on the former rokepxa patch by Cédric), that fixes problems with the ARM Linux patch and adds support for the specific features of RokEPXA.

We activated floating-point emulation at the kernel level, because when we tried to do it at the compiler level we ran into trouble. This method is also more optimal (saves code) and will scale better if we someday end up having a floating-point coprocessor in VHDL.

4.2 NFS Support

By having NFS support, we are able to transfer data between the RokEPXA and the desktop developement machine at a much faster rate than with the parallele port. We also no longer need to flash the ROM each time we change or create files, because we can access them remotely.

The NFS uses the Ethernet support provided by the (rokepxa_extcam3d.sbi) coprocessor. The only problem with NFS is that we cannot have NFS and another coprocessor loaded at the same time.

4.3 Native-compiler

We created a native compiler "by accident". We needed the ability to have dynamically linked libraries. At first we thought that this is done by "ld". It turns out that "ld" is just the linker, the dynamic loader is in the kernel (and all it requires is that the LD_LIBRARY_PATH is set correctly).

4.4 Kaffe JVM

http://www.kaffe.org/

We worked with Kaffe JVM (ver 1.1.3). The main reason is because it is open source (Sun's code isn't) and there might be a need to modify it. It also scales well with our small system.

5 Research

This section documents the research we did. It should provide you with all the knowledge needed to understand our final implementation.

5.1 Inside Java And It's Virtual Machine

Java is a very rich programming language with many unique[11] features. When compared to other languages, Java has a layer of indirection called the JVM. This makes it easy to run the same Java code anywhere. The JVM is specifically designed for the Java programming languages, but can of course run any language (as long as someone writes the compiler).

A Java program is a collection of class files. Class files are created by compiling the java source code. The class file contains a constant pool and Java bytecode, which is the basic instructions that the JVM understands. The bytecode is similar to assembly, it is designed around a stack-based architecture.

The JVM is defined by "The Java Virtual Machine Specification". Besides running bytecode, it also performs verifications (static and dynamic) and garbage collection.

5.1.1 Class Loader

Code in Java is dynamically linked. Each time a class is referenced, a class loader is called to locate and load the bytecode. One of the interesting features of Java is the ability to write a custom class loader.

The JVM has a built-in primordial class loader, which it uses to load itself. Once the JVM has started, every time a class needs to be loaded, it is done with one of the class loader's written in Java (usually one of java.lang.ClassLoader, java.net.URLClassLoader, sun.applet.AppletClassLoader, etc.).

The tasks of the class loader are:

1. Locate the bytecode (on the disk or on the network).
2. Call defineClass to check that the bytecode is properly formatted. It also detects ClassCircularityError (when a class is itself the superclasss of one of it's superclasses).
3. Load the superclass (and the superclass of the superclass, and so on).

[11] Now being copied by others like .Net

4. Calls resolveClass so that the bytecode can be verified.

5.1.2 Bytecode

Java bytecode uses a stack-based architecture. So, for example, adding 2+3 requires you to do the following:

```
bipush 2    /* Push the number 2 on the stack. */
bipush 3    /* Push the number 3 on the stack. */
iadd        /* Add the stack's top two elements, the arguments
               are poped and the result is on the stack. */
```

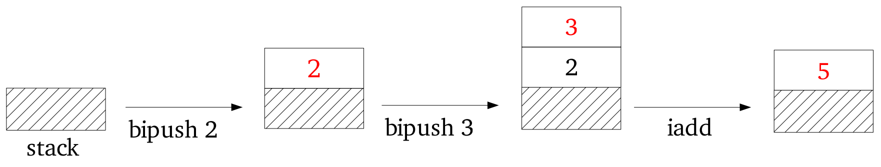


Figure 1: adding 2+3

The bytecode also makes extensive use of strings. These strings are stored in the constant pool and are references by their entry number. For example an object creation or method invokation is done by it's name (as a string). Example of a System.out.println("Hello World");

```
/* Get the out field from System. */
getstatic java/lang/System/out Ljava/io/PrintStream;

/* Push the constant string on the stack. */
ldc ''Hello World''

 /* Call the println method. */
invokevirtual java/io/PrintStream/println (Ljava/lang/String;)V
```

There are several types of method invokation:

- invokevirtual: invokes a non-static (instance) method.
- invokestatic: invokes a static method.
- invokespecial: used for constructors.
- invokeinterface: used for interfaces.

Methods are identified by their class name, method name and signature. A method can have several attributes:

- public: This method is available to all other classes.
- private: This method may be accessed only from within this class.

- protected: This method may be accessed from any class in the same package or any class that is a subclass of this class.
- static: This method is static.
- final: This method may not be overridden in subclasses.
- synchronized: The JVM will obtain a lock on this object before invoking the method. If this method is static, the JVM will obtain a lock on the Class object.
- native: This method is implemented in native code.
- abstract: This method has no implementation.

Two methods in a given class may not have the same name and signature (except if the argument part of the signature differ, a feature known as method overloading). It is interesting to note that even if the attributes differ (for eg: a static vs a non-static method) the name and signatures must differ.

The signature is composed of two parts, the argument types and the return type. The argument types is a list (enclosed in parenthesis). The signature has the following format:

$$(type_1 type_2 type_3 ...)type_{return}$$

See Table 1 for the different formats of *type*. For example the signature of **void main(String argv**[]**)** is $([Ljava/lang/String;)V$

Table 1: Signature formats

Type	**Descriptor**
array of *type*	[*type*
byte	B
boolean	Z
char	C
double	D
float	F
int	I
long	J
reference to *classname*	L*classname*;
short	S
void	V

5.1.3 BCEL

http://jakarta.apache.org/bcel/index.html

BCEL[12] is a library that is part of the Apache/Jakarta project. This library provides a way to manipulate Java bytecode, providing an amazing amount of power and flexibility (when I discovered this library I felt the same way as when I discovered Java reflexion a couple of years ago: I realized this library provides a whole new dimension to Java).

BCEL can be used after compile time (on *.class* files) or at load time (by writing a custom class loader). It has been used in many projects, including:

- compiler backends
- extending the Java language (multi-inheritence, parameterized classes[13], preprocessor, custom bytecode, etc...)
- performing optimizations (like peep-hole)

BCEL includes a nice feature BCELifer, which takes a *.class* file and generates the Java code that will generate the *.class* file. This tool proves useful for testing BCEL and developement.

The BCEL libary (probably for historical reasons) has two parts: one set of classes to read bytecode and one set to modify and generate bytecode. This can be quite confusing at first.

5.1.4 Java Native Interface

The JVM provides a standard way of calling external code written in C (or any other language). The C code must be compiled as a shared library, and a JNI wrapper class must be created (that will wrap calls to Java methods into the shared library).

JNI also provides a way for the native method to create, access and update Java objects.

5.1.5 Garbage Collection

One of the features of Java is to have automatic memory management. This means that the programmer doesn't need to allocate and free memory, it is handled automatically by the JVM (by using a technique known as garbage collection).

Basically a garbage collector finds what objects cannot be referenced by a program and reclaims the storage used by those objects.

At first, we thought that garbage collection could be an issue for our project. If the coprocessor is running in parallel with the JVM, then the

[12]Byte Code Engineering Library
[13]Java 1.5 features this

JVM doesn't know what objects are still being used in the coprocessor. This can cause the garbage collector to delete something the coprocessor might need. Some implementations of garbage collectors also move data arround, this can also create problems if a pointer to the data was passed to the coprocessor.

By using JNI, we were able to ignore this aspect in our projects. JNI keeps track of what references are being used by the native application.

5.1.6 Java Hot Spot

The nature of the Java language and it's Virtual Machine has brought many people to do research in the field of "Java Hot Spot"[14]. The idea is to dynamically analyse a Java program execution (know which methods are invoked most often and where time is being spent), and to take optimization decisions as the program executes. It is also possible to recompile a class and dynamically replace it (HotSwap Class File Replacement).

Unfortunately, Kaffe doesn't have anything similar yet.

5.1.7 Security Issues

Java security is very complex. To put things in a simple way: by using a custom class loader or native methods, the security system is basically doomed. If security is really a matter, there are ways to digitally sign the class files in order to specify what is to be trusted. It is also possible to specify the security level of custom generated code (such as code generated by BCEL).

5.2 IDEA

IDEA is a cryptography algorithm. It takes 64 bit plaintext block and encrypts it using a 128 bit key. It is a symetric algorithm, so the same key is used to decrypt the coded text.

We used IDEA to compare the time it takes to encrypt data with a software implementation (Java and C) as opposed to a hardware implementation (VHDL).

5.3 Generating VHDL From Java

In theory it is possible to automatically convert Java to VHDL, probably with under some constraints. I did not have enough time to delve into this topic, but it can offer interesting possibilties to our projects. This is definately a future work.

[14]HotSpot is actually a trademark of Sun, but I couldn't find any generic term

5.4 Different Solutions To My Project

There are different ways in which I can get Java to invoke the coprocessor. I opted for a solution that is portable accross JVM's. This was not a requirement I had, but it's an elegant method.

- **Modifying the JVM at invokation level**
  Since we have the source code of Kaffe, we could modify the invokation method and take a decision whether to execute the coprocessor or normal Java code.

- **Catching "no such method" exceptions**
  If we assume that methods implemented in the coprocessor don't have Java implementations, then we could catch "no such method" exceptions and try to run the coprocessor instead. One problem with this solution is how to continue execution from the instruction that caused the exception.

- **Creating a custom class loader**
  I opted for this solution. The idea is to create a custom class loader. This class loader is used to load all the classes. If the current class it is loading has methods implemented in the coprocessor, the class loader will modify the bytecode and generate a method invokation (this solution uses Christophe's code). If the class doesn't need any special treatment, the default class loader is used. This method also has the advantage of being able to simultaneously have Java code and a coprocessor allowing the application user to choose which method to use.

  Figure 2 illustrates how VMWClassLoader (the custom class loader) is loaded and how it takes over the loading thereafter.

- **Modifying .class files**
  A similar method to the above, would be to patch the `.class` files before running the JVM. The advantage of this method is that it doesn't require a custom class loader. It does have the drawback of being less dynamic than by having a custom class loader.

- **Debugging framework**
  Some JVM have debugging support, which means you can write code that interceps method invokations. This solution is probably the best of all, because it also allows an extension to Java HotSpot technology. We could even imagine extending HotSpot so that the JVM can receive information about the coprocessor (method size in gates and timing).

  Unfortunately I thought about this solution near the end of my project. As mentioned previously, Kaffe doesn't support such a feature.

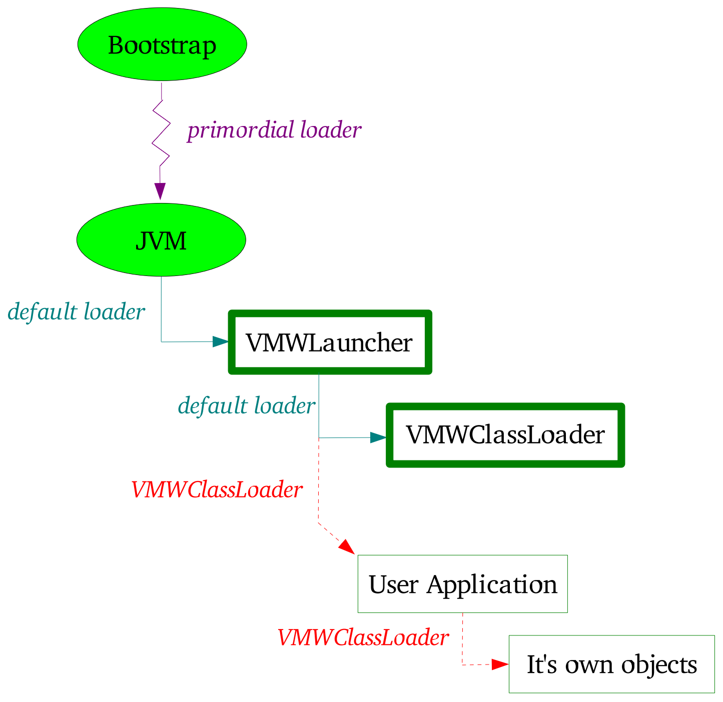
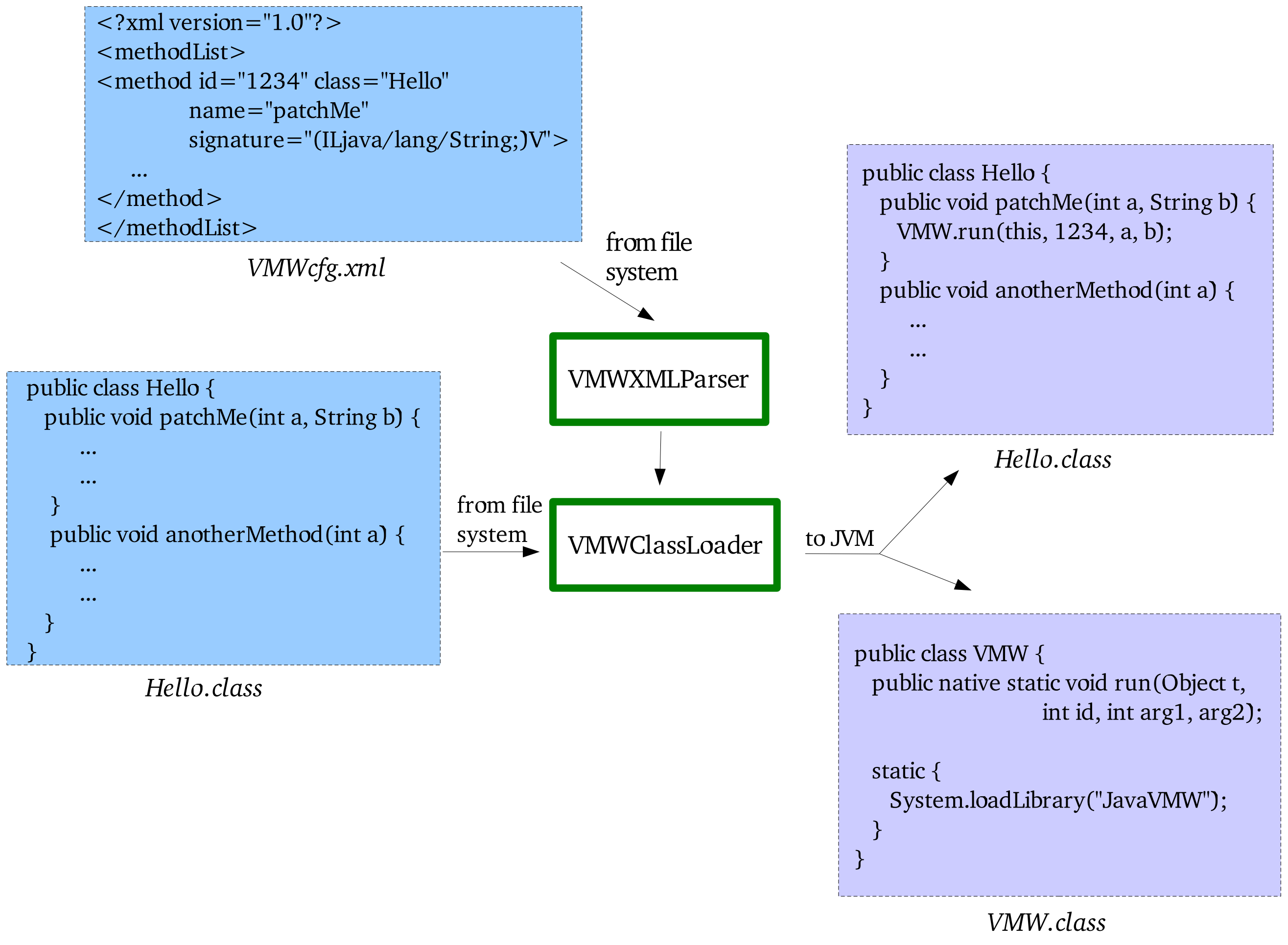


Figure 2: The different class loaders used at each stage



```
<?xml version="1.0"?>
<methodList>
<method id="1234" class="Hello"
            name="patchMe"
            signature="(ILjava/lang/String;)V">
     ...
</method>
</methodList>
```

```
public class Hello {
  public void patchMe(int a, String b) {
      ...
      ...
  }
  public void anotherMethod(int a) {
      ...
      ...
  }
}
```

```
public class Hello {
  public void patchMe(int a, String b) {
    VMW.run(this, 1234, a, b);
  }
  public void anotherMethod(int a) {
      ...
      ...
  }
}
```

```
public class VMW {
  public native static void run(Object t,
                        int id, int arg1, arg2);

  static {
    System.loadLibrary("JavaVMW");
  }
}
```

Figure 3: How VMWClassLoader handles the user classes and generates VMW.class

6 Implementation

As discussed above, I used a custom class loader *VMWClassLoader.java* to load all the classes. If the class loader needs to, it can delegate the loading to the default loader.

6.1 XML Config File

We used an XML (*VMWcfg.xml*) based configuration file. One could argue that the coprocessor can't be used in a transparent way if such a config file is necessary. I will not discuss this issue here, as it was necessary for me to find a simple way to know which methods are available as coprocessors and for Christophe's code to know which *.map* file corresponds to each method.

We opted for a XML format, because this allows us to use the same config file for both the projects. You might think this choice is bad because we are working with embedded systems. It turns out that the config file is parsed only once (actually twice, once in my code and once in Christophe's) and the format is simple enough so we can write very efficient parsers if need be.

Here is what it looks like:

```
<?xml version="1.0"?>
<methodList>
  <method id="1" class="AnotherClass"
          name="patchMe" signature="([S[S[S)V">
    <map>test.sni</map>
    ...
  </method>
</methodList>
```

There can be many other things inside the `<method>` and `/method` tags, but I only use the attributes of the `<method>` tag (id, class, name and signature).

I used `javacc` (https://javacc.dev.java.net/) to generate a parser (VMWXML-Parser.jj).

The information I retrieve from the configuration file is saved in LinkedList of *ClassIdMethodSignature.java* and *IdMethodSignature.java* objects (these classes are only wrapper classes).

See § 9.1.11 for the dtd of our config file.

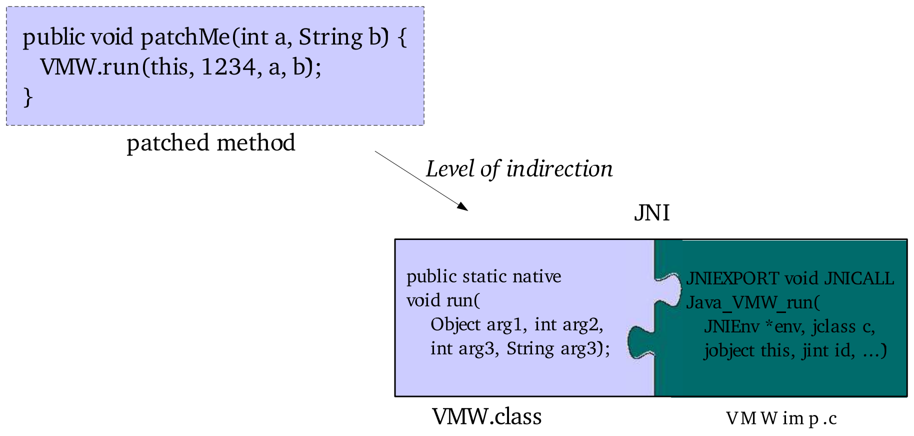
6.2 Interfacing With JNI

The file *VMWimp.c* contains the C code written by Christophe that must be called from Java (this code gets compiled into a library called *libJavaVMW.so* on Linux). This code loads the coprocessor and runs it. The function to be called is *Java_Vim_run*:

JNIEXPORT void JNICALL Java_VIM_run(JNIEnv *env, jclass cls, jobject this, jint id, ...).

The jobject is a pointer (it's a real pointer; remember this is a native function) to the object that is calling the coprocessor. This is needed if the coprocessor wants to do callbacks to the JVM. Since we have a level of indirection (see Figure 4), the C code is actually called from a static Java context (the VMW.class wrapper); but we are interested in the object that contains the static call (the patched method). The value is NULL if the context is static.

It could have been possible to remove this level of indirection, but I think this design is better: less work is required at class load time and it is the only way to do things if you want to hot-swap classes in the future.



```
public void patchMe(int a, String b) {
  VMW.run(this, 1234, a, b);
}
```

```
public static native
void run(
    Object arg1, int arg2,
    int arg3, String arg3);
```

```
JNIEXPORT void JNICALL
Java_VMW_run(
   JNIEnv *env, jclass c,
   jobject this, jint id, ...)
```

Figure 4: How the patched method invokes the library

At first we thought of passing the class name, method name and method signature to the native library. This worked, but presented a performance problem: for the native code to be able to manipulate the strings, it must perform callbacks to the JVM. Since both our projects share the same configuration file, we opted for an id that is attributed to each `<method>` tag in the config file. The reason the native code needs to know which method is calling it, is primarly so that it can locate the right *.map* file.

As you might have noticed, the C code takes a variable number of arguments. Unfortunately Java doesn't have variable number of arguments, but it has method overloading (methods with the same name but different signatures). On the other hand, C doesn't have method overloading. What JNI does is it maps every method called *run* in the class *VMW.class* to the C function *Java_VMW_run*. This means we must dynamically generate *VMW.class* (see Figure 3); using the XML config file, we will generate a method declaration for each possible signature. (VMW.class is actually generated using the VMWSkel.java skeleton file).

Here is an example of a *VMW.java* file. Of course this file never actually exists on the disk:

```
public class VMW {
  public native static void run(Object t, Int id, int a, String b);

  static {
    System.loadLibrary(VMWConstants.libFile);
  }
}
```

At first we had thought of using an array of Objects instead of the variable number of arguments. This has the main advantage of making the C code simpler because working with variable arguments is a *pain*. The array of Objects presents some problems though: primitives (int, byte, char, etc.) must be wrapped in an Object and then unwrapped because the native code will handle primitives differently than Objects (the native code can manipulate primitives, but it needs to make callbacks to use Objects). The other problem is related to performance, the C code must make multiple callbacks to the JVM in order to manipulate the array of Objects. It is interesting to note that Java 1.5 supports variable number of arguments, but doesn't solve the problem of wrapping and unwrapping primitives.

6.3 Launcher Class

Creating a custom class loader presents some problems. In order to be able to load all the classes with the custom class loader, we need to replace the default loader. Unfortunately this cannot be done using an environement variable when invoking the JVM. One way to solve this problem is to modify the class package that is included in Kaffe. There is another more elegant way of getting arround: by using a launcher class (*VMWLauncher.java*).

The launcher class will be responsible for loading the user's main class using the custom class loader and then invoking the **static void main(Object[] argv)** method. The way the custom class loader is designed, it will be called on any subsequent class loading (see Figure 2).

The launcher class uses reflection to check that the user's main class has a static main method. It also shifts the argv array by one element (so the user's main class has "no idea" there was a level of indirection).

It turns out that Kaffe has a bug, it will not allow the custom class loader to inherit the classpath from the default loader. The work arround involves two steps: first I declared a constant (VMWConstants.kaffebug) which contains the path where the user's class files reside. The second step is a fix at the custom class loader (see below).

6.4 Custom Class Loader

The custom class loader (*VMWClassLoader.java*) extends URLClassLoader which means it can handle .jar & .zip archives too.

The first thing the loader does is parse the XML config file.

The loader contains code from Kaffe's URLClassLoader; due to the class loader architecture, the *VMWClassLoader.java* cannot call URLClassLoader's *myFindClass* or otherwise subsequent class loading won't call *VMWClassLoader*. (the URLClassLoader's findClass calls defineClass).

The loader also contains code from BCEL due to a misdesign in the library (you can't directly create a *ClassGen* from an array of bytes).

BCEL containts a class loader, but it is not suitable for my project, because it only loads classes with specific name patterns.

Basically the custom loader has four cases:

- **The class is already loaded.** We ask the JVM (findLoadedClass) if the class is already loaded. If it is, we just return what findLoadedClass returned.

- ***VMW.class* needs to be dynamically created.** We iterate through the linkedlist (generated by the XML parser) and create a method declaration for every possible signature. We mustn't generate two declarations if two different methods have the same signature.

- **The class is listed in the XML config file.** We load the class and patch each of the methods listed in the config file. We replace the method's body by an invokation to:
  VMW.run(this, id, arguments_of_the_original_method). The first argument is NULL if the method being patched is static. If the method is synchronized, it will generate the monitorenter and monitorexit code.

- **The class is a normal class (none of the above 3 cases).** In the case of a normal class, the custom class loader must still load it manually (and not delegate to the system loader) in order to maintain control over what the normal class will load. If the custom class loader can't find the class (this can happen in Kaffe due to the bug discussed above), then the default System loader will be called. (This should be fixed as soon as the bug in Kaffe is solved).

It is important to notice that the original arguments are numbered from 1 to N for non-static methods (because argument 0 is 'this'), but from 0 to N-1:

```
public static void Hello(int a) {
  /* in this method, a is argument 0,
     it can be accessed with ILOAD(0) */
}

public void Hello2(int a) {
  /* in this method, a is argument 1,
  it can be accessed with ILOAD(1) */
}
```

Note: I discovered and fixed a bug in BCEL that prevents it from generating the right code for native methods. The fix has been accepted for the next release of BCEL, but until then, you must use my patched version (bcel-5.1b).

7 Limitations / Improvements

- Methods that are patched don't return anything. Currently this is not really a limitation, since we don't have a limit on the number of arguments (and you can always replace a return argument by an extra input argument). This might become a limitation in the future. We made this decision in order to simplify various parts of the code (VMWClassLoader doesn't need to generate different return instructions, the native method doesn't need to handle return values, etc.).
- We have used file names such as VMWClassLoader, VMWXMLParser, etc. It would be a wise idea to declare a package name for all these files and avoid having name space collisions.
- The config file is actually parsed twice, once in the Java code and once in the native code. Altough each parser uses different information, it is possible to parse the file only once.
- The class files are also parsed twice. Once by BCEL before patching, and once by the JVM (VMWClassLoader calls defineClass which takes an array of bytes). This is a negligeable thing.
- It would be interesting to be able to reload classes based on real-time statistics (HotSpot). This could be an interesting future work.
- It would be interesting to research how to convert Java code into VHDL automatically. This can also be an interesting future work.

8 Conclusions and Future work

The main goal of my project has been reached. I am successfuly able to call coprocessors from Java, without modifying or recompiling the source code. The work I did can also be used to call C code from Java (thus simplifying the coding of native libraries for Java). We could imagine that depending on the hardware available or security policies, a native code or Java code could be executed.

The overhead of the custom class loader is very negligible compared to the time gained in the coprocessor. The time required to load my code and bcel is very little as compared to the time required to load the JVM. The patching of methods is very fast, and is done only once. Currently the main problem is in the time required to load the coprocessor, but this time can be "hidden" by having the processor do something else meanwhile.

This project involved very interesting aspects of Java programming. I gained detailed insight at the internals of a JVM. I also spent a lot of time setting up the development environment. This process was time consuming and sometime frustrating, but it clearly highlighted how difficult and complex open source projects can sometimes be. By using hardware combinations that not a lot of people have tried, we ran into new problems on a day to day basis.

There are two open questions that I wish I would have had more time to work on: how to replace a class in the JVM in real-time and how to generate VHDL from Java bytecode. Both these problems seem complex and could very well be an entire project on their own, but these are the interesting problems that will provide new properties to my existing project.

An interesting possiblity I did not cover in this report, is that the coprocessor can call the JVM (by using JNI). We did get this working (by displaying a "Hello World" back from the coprocessor), but we didn't explore the possiblities of such a "reverse" system.

9 Appendix

9.1 Code

9.1.1 Organization

Here is the list of files and what they do:

- **VMWXMLParser.jj** This is the javacc file that will generate the XML parser. Once compiled by javacc, it will generate the following files: VMWXMLParser.java, TokenMgrError.java, ParseException.java, Token.java, SimpleCharStream.java.
- **VMWClassLoader.java** This is the custom class loader that does all the work.
- **VMWLauncher.java** The launcher used to set the custom class loader.
- **VMWConstants.java** File containing all the constants used in the program.
- **ClassIdMethodSignature.java** This file is just a wrapper.
- **IdMethodSignature.java** This file is just a wrapper.
- **VMWSkel.java** Used to generate VMW.class on the fly.
- **VMW.class** Generated on the fly, never exists as a file.
- **VMWcfg.xml** User configuration file.
- **libJavaVIM.so** Native library.
- **VWMXMLcfg.dtd** DTD for config file.

9.1.2 Compiling & Running

Compiling the source code is straight forward, all you need is the bcel library:

```
javacc VMWXMLParser.jj
javac -classpath ‘‘bcel-5.1b.jar’’ *.java
```

To run some code (eg: Hello) using my code:

`java -cp ‘‘bcel-5.1b.jar:.’’ VMWLauncher Hello` *parameters for Hello*

9.1.3 VMWXMLParser.jj

Listing 1: VMWXMLParser.jj

```
/*
 * Created on Jan 14, 2004
 *
 * Javacc file, to parse the xml config file.
 * I didn't implement the entire dtd, only what
 * interests me. The goal is to be the most
 * optimized possible.
 *
 * When parseFile is called (from VMWClassLoader),
 * the static LinkedList config will contain ClassIdMethodSignature
     objects.
 *
 */

/**
 * @author Alok Menghrajani
 *
 */

options {
  STATIC = true;
  DEBUG_PARSER = false;
}

PARSER_BEGIN(VMWXMLParser)

import java.io.FileReader;
import java.util.*;

public class VMWXMLParser {

  public static LinkedList config;

  /* For testing purpose only */
  public static void main(String args[]) throws Exception {
    FileReader in = new FileReader(args[0]);
    VMWXMLParser parser = new VMWXMLParser(in);
    parser.XMLDocument();
    System.out.println("Ok");
  }

  public static void parseFile(String file) throws Exception {
    /*
      This is the method that will be called from VMWClassLoader.
    */
    FileReader in = new FileReader(file);
    VMWXMLParser parser = new VMWXMLParser(in);
    parser.XMLDocument();
  }
}

PARSER_END(VMWXMLParser)

SKIP : {
  " " | "\r" | "\t" | "\n"
}

TOKEN : {
    < LT: "<" >
  | < GT: ">" >
  | < EQUAL: "=" >
  | < SLASH: "/" >
  | < QUOTE: "\"" >
  | < XML_S: <LT> "?" >
  | < METHODLIST_S: <LT> <METHODLIST> <GT> >
  | < METHODLIST_E: <LT> <SLASH> <METHODLIST> <GT> >
  | < METHOD_S: <LT> <METHOD> >
  | < METHOD_E: <METHOD> <GT> >
  | < ID: "id" >
  | < CLASS: "class" >
  | < NAME: "name" >
  | < SIGNATURE: "signature" >
  | < #METHOD: "method" >
  | < #METHODLIST: "methodList" >
  | < ANY: ~[] >
        | < VAL: <QUOTE> (~["\""])* <QUOTE> >
}


JAVACODE

/*  Flush until a '>' is encountered. Useful when you want to
  skip a tag that you don't care about. */
void flush() {
  while ((getNextToken()).kind != GT)
    ;
}

String string() : {} {
  <VAL> {
    String s = token.image;
    return s.substring(1, s.length()-1);
  }
}

void XMLDocument() throws Exception : {
  String cl, n, sig;
  short id;
} {
  {
    /* Initialize the Vectors */
    config = new LinkedList();
  }
  <XML_S> {flush();}  /* skip <?xml version ?> */
  <METHODLIST_S>

  (<METHOD_S>
    <ID><EQUAL>{id=Short.parseShort(string());}
    <CLASS><EQUAL>{cl=string();}
    <NAME><EQUAL>{n=string();}
    <SIGNATURE><EQUAL>{sig=string();}
    {
      /* Search to linkedlist to see if I don't already have this
          class */
      ListIterator i = config.listIterator(0);
      boolean flag = false;
      while (i.hasNext()) {
        ClassIdMethodSignature cims = (ClassIdMethodSignature) i.next
            ();
        if (cims.className.equals(cl)) {
          /* Add the id, method and signature only.
             (Perform check to see that this combination of
              method/signature doesn't already exist)
          */
          ListIterator j = cims.idMethodSignature.listIterator(0);
          while (j.hasNext()) {
            IdMethodSignature ims = (IdMethodSignature) j.next();
            if (ims.method.equals(n) && ims.signature.equals(sig)) {
              // Check that method returns void !
              // xml file is bad.
              // Warning: I'm not going to check
              // for something like this (totally illegal):
              // int add(int, int) and
              // String add(int, int) (two different
              // signatures, but collision !)
              throw new RuntimeException("Bad XML file");
            }
          }
          cims.idMethodSignature.add(new IdMethodSignature(id, n, sig)
              );
          flag = true;
        }
      }
      /* Add the class */
      if (!flag) {
        ClassIdMethodSignature cims = new ClassIdMethodSignature(cl);
        cims.idMethodSignature.add(new IdMethodSignature(id, n, sig));
        config.add(cims);
      }
    }
  <GT>
  { while((getNextToken()).kind != METHOD_E); } /* skip what's under
      the <method> tag */
  )*

  <METHODLIST_E> <EOF>
}
```

9.1.4 VMWClassLoader.java

Listing 2: VMWClassLoader.java

```
/*
 * Created on Jan 13, 2004
 *
 * The 'core' of my project.
 * This is a special class loader which performs the following tasks:
 *
 * - reads XML config file.
 *
 * - loads classes, that aren't listed in XML, normally (using
     VMWClassLoader if the
 *   class can be found, otherwise using the system loader if the
     class can't be found
 *   (should only happen in Kaffe due to a little bug)).
 *
 * - loads classes specified in config file (classes that contain
 *   optimized method) and 'patches' the optimized methods by calling
    VMW.run
 *
```

```java
 * - creates special class VMW (native library wrapper),
 *   adding one run method for each possible signature (by looking in
 *   the config file). This class is created from 'scratch' using
     VMWSkel.
 *
 * VMW's run methods take 2 extra parameters:
 * this = a reference to the caller (or null if context is static).
 * id = id allowing the native code to find the entry in the config
      file.
 */

/**
 * @author Alok Menghrajani
 *
 */

import java.net.*;
import java.io.*;
import java.util.*;

import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

public class VMWClassLoader extends URLClassLoader {

  static {
      if (VMWConstants.debug)
    System.out.println("VMWClassLoader loading...");
      try {
    /* Parse the XML config file. */
    VMWXMLParser.parseFile(VMWConstants.configFile);
    System.out.println("VMWClassLoader ready.");
      } catch (FileNotFoundException e) {
    System.out.println("XML config file "+VMWConstants.configFile +"
        not found.");
    System.exit(-1);
      } catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
    System.exit(-1);
      }
  }

  public VMWClassLoader(URL[] url) {
      super(url);
  }

  public Class loadClass(String name) throws ClassNotFoundException {
    return loadClass(name, false);
  }

  public Class loadClass(String name, boolean resolve) throws
      ClassNotFoundException {
    /* This method is called to load a class.
     *
     * There are 4 possibilities:
     * 1) The class is already loaded
     *    -> I test this using findLoadedClass.
     * 2) The class is listed in the XML config file
     *    -> I load it using myFindClass and then
     *       patch it using BCEL.
     * 3) The class to be loaded is VMW (java native class)
     *    -> Build it from scratch.
     * 4) The class is a normal/system class
     *    -> I load it using myFindClass or findSystemClass.
     *
     */

    Class result = findLoadedClass(name);
    if (result != null) {
        /* We are in case 1 - the class is already loaded. */
        if (VMWConstants.debug)
      System.out.println("Already loaded: " + name);
        return result;
    }

    /* Check to see if it's VMW */
    if (name.equals("VMW")) {
        if (VMWConstants.debug)
      System.out.println("Create VMW");
        return createVMW();
    }

    /* Check the VMWXMLParser.config LinkedList to see if
       it's a class that needs to be patched */
    ListIterator i = VMWXMLParser.config.listIterator(0);
    while (i.hasNext()) {
        ClassIdMethodSignature cims = (ClassIdMethodSignature) i.next
            ();
        if (cims.className.equals(name)) {
      if (VMWConstants.debug)
          System.out.println("Special class: " + name);
      byte[] bytes = myFindClass(name);
      result = patchClass(cims, bytes);
      if (resolve) {
          resolveClass(result);
      }
      return result;
        }
    }

    /* We are in case 2 - the class is a system/normal class */
    try {
        byte[] bytes = myFindClass(name);
        if (VMWConstants.debug)
      System.out.println("Normal Class: " + name);
        Class cl = defineClass(name, bytes, 0, bytes.length);
        if (resolve) {
      resolveClass(cl);
        }
        return cl;
    } catch (Exception e) {
        if (VMWConstants.debug)
      System.out.println("System Class: " + name);
        return findSystemClass(name);
    }
  }

  private byte[] myFindClass(String name) throws
      ClassNotFoundException {
    /* Mostly inspired/pasted from Kaffe sources. */
          /* It seems Kaffe imposes a 1024 max file size for class */
```

```java
    URL url = findResource(name.replace('.', '/') + ".class");
    if (url == null) {
      throw new ClassNotFoundException(name);
    }
    try {
      InputStream in = url.openStream();
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      byte[] buf = new byte[1024];
      for (int r; (r = in.read(buf)) != -1; ) {
        out.write(buf, 0, r);
      }
      in.close();
      buf = out.toByteArray();
      return buf;
    } catch (IOException e) {
      throw new ClassNotFoundException(name + ": " + e);
    }
  }

  private JavaClass createJavaClass(String classname, byte[] bytes) {
    /* Inspired by BCEL sources. */
    JavaClass clazz = null;
    try {
      ClassParser parser = new ClassParser(new ByteArrayInputStream(
          bytes), "foo");
      clazz = parser.parse();
    } catch(Throwable e) {
      e.printStackTrace();
      return null;
    }

     ConstantPool cp = clazz.getConstantPool();
     ConstantClass cl = (ConstantClass)cp.getConstant(clazz.
         getClassNameIndex(), Constants.CONSTANT_Class);
     ConstantUtf8 name = (ConstantUtf8)cp.getConstant(cl.getNameIndex
         (), Constants.CONSTANT_Utf8);
    name.setBytes(classname.replace('.', '/'));
     return clazz;
  }

  private Class patchClass(ClassIdMethodSignature cims, byte[] bytes)
       {
    /* Patches Class cl, so that all invokations of methods
     * that appear in XML config file are rerouted to VMW.run
     *
     * Here are the major steps performed:
     * 1) Find every method (listed in cims) in Class
     * 2) Convert signature from String to Type[].
     * 3) Replace body by a call to VMW.run(this (or null),
     *    id, param1, param2, ...)
     */

    JavaClass jc = createJavaClass(cims.className, bytes);
    ClassGen clazz = new ClassGen(jc);
    Method methods[] = clazz.getMethods();
    for (int i=0; i<methods.length; i++) {
        /* Search in cims.methodSignature LinkedList for
         * a match with methods[i].getName() and methods[i].
             getSignature().
         */
        ListIterator j = cims.idMethodSignature.listIterator();
        while (j.hasNext()) {
```

```
        IdMethodSignature ims = (IdMethodSignature)j.next();
        if (ims.method.equals(methods[i].getName()) &&
            ims.signature.equals(methods[i].getSignature())) {
            /* Set the flag, so we can show a warning if
             * a method isn't present in the class but
             * is in the XML config file.
             */
            if (ims.flag) {
          throw new RuntimeException("flag already set");
            }
            ims.flag = true;
            /* Patch this method */
            Method patch = patchMethod(clazz, methods[i], ims.id);
            clazz.replaceMethod(methods[i], patch);
        }
          }
      }

      /* Check the flags */
      ListIterator i = cims.idMethodSignature.listIterator();
      while (i.hasNext()) {
          IdMethodSignature ims = (IdMethodSignature) i.next();
          if (!ims.flag && VMWConstants.debug) {
        System.out.println("Warning: method "+cims.className+"."+ims.
            method+"{"+
               ims.signature+"} not found in class file but used in
                   config file.");
          }
      }

      /* Register new class */
      byte[] output = clazz.getJavaClass().getBytes();
      Class cl = defineClass(cims.className, output, 0, output.length);
      return cl;
    }

    private Method patchMethod(ClassGen clazz, Method o, short id) {
      /* Replace the body of this method by a call to VMW.run */
      ConstantPoolGen cpg = clazz.getConstantPool();
      MethodGen mg = new MethodGen(o, clazz.getClassName(), cpg);

      /* We will call a run method of VMW that has the same signature.
       * We still need to create a new signature, because VMW will take
       * a few extra parameters (this, id)
       * */
      Type[] arguments = Type.getArgumentTypes(o.getSignature());
      Type[] newtype = new Type[2+arguments.length];
      newtype[0] = Type.OBJECT; // reference to caller
      newtype[1] = Type.INT;
      for (int i=0; i<arguments.length; i++) {
        newtype[i+2] = arguments[i];
      }
      String newsignature = Type.getMethodSignature(Type.VOID, newtype);
      if (VMWConstants.debug)
          System.out.println("NEW SIGNATURE="+newsignature);

      int vmwrun = cpg.addMethodref("VMW", "run", newsignature);
      if (vmwrun==-1) {
        System.out.println("Fatal error in patchMethod (vmwrun=-1)");
        System.exit(-1);
      }
    /* Start creating new instructions */
    InstructionList il = new InstructionList();

    /* Check if this method is in a static context. */
    /* in static context, var 0 = param 1, var 1 = param 2, etc..
       in instance context, var 0 = this, var 1 = param 1, etc...
    */
    int first_param;

    if (o.isStatic()) {
        il.append(new ACONST_NULL());
        first_param = 0;
    } else {
        il.append(new ALOAD(0));
        first_param = 1;
    }

    il.append(new SIPUSH(id));

    for (int i=0; i<arguments.length; i++) {
      il.append(new ILOAD(i+first_param));
    }
    il.append(new INVOKESTATIC(vmwrun));
    il.append(new RETURN());

    mg.setInstructionList(il);
    mg.removeLineNumbers();
    mg.setMaxStack();

    return mg.getMethod();
  }

  private Class createVMW() {
    /* Create VMW Class using VMWSkel */
    ClassGen clazz = new ClassGen(Repository.lookupClass("VMWSkel"));
    clazz.setClassName("VMW");

    ConstantPoolGen cpg = clazz.getConstantPool();
    InstructionFactory factory = new InstructionFactory(clazz, cpg);

    InstructionList il = new InstructionList();
    /* Create a method declaration for each possible signature */
    LinkedList signatures = new LinkedList();
    ListIterator i = VMWXMLParser.config.listIterator(0);
    while (i.hasNext()) {
      ClassIdMethodSignature cims = (ClassIdMethodSignature) i.next();
      ListIterator j = cims.idMethodSignature.listIterator(0);
      while (j.hasNext()) {
        IdMethodSignature ims = (IdMethodSignature) j.next();
        /* Check to see if we haven't already handled this signature
            */
        if (!signatures.contains(ims.signature)) {
          signatures.add(ims.signature);
          Type[] arguments = Type.getArgumentTypes(ims.signature);
          Type[] newtype = new Type[2+arguments.length];
          newtype[0] = Type.OBJECT; // reference to caller
          newtype[1] = Type.INT;    // id
          for (int k=0; k<arguments.length; k++) {
            newtype[k+2] = arguments[k];
          }

          String[] strings = new String[newtype.length];
          for (int k = 0; k<strings.length; k++) {
            strings[k] = "arg"+k;
          }
          /* We can assume the return type is void */
          if (VMWConstants.debug)
              System.out.println("ADDING: " + ims.signature);

          MethodGen mg = new MethodGen(Constants.ACC_PUBLIC |
              Constants.ACC_STATIC | Constants.ACC_NATIVE, Type.VOID,
            newtype, strings, "run", "VMW", il, cpg);

          clazz.addMethod(mg.getMethod());
        }
      }
    }

    /* Register new class */
    byte[] output = clazz.getJavaClass().getBytes();
    Class cl = defineClass(clazz.getClassName(), output, 0, output.
        length);
    return cl;
  }
}
```

9.1.5 VMWLauncher.java

Listing 3: VMWLauncher.java

```
/*
 * Created on Jan 13, 2004
 *
 * This is a wrapper to launch Java applications (name passed in arg
    [0]).
 * The goal is the have AlokClassLoader load the classes for this
 * Application.
 *
 * The remaining arguments will be passed to the main Java application
    .
 *
 */

/**
 * @author Alok Menghrajani
 *
 */

import java.lang.reflect.*;
import java.net.*;

public class VMWLauncher {

  public static void main(String args[]){
    if (args.length==0) {
      System.out.println("Usage: java AlokLauncher appName extraargs")
          ;
      System.exit(-1);
    }

    // Check to see if there are any arguments to be
    // passed to the main Java app.
    String args2[] = new String[0];

    if (args.length>1) {
      args2 = new String[args.length-1];
      for (int i=1; i<args.length; i++) {
        args2[i-1] = args[i];
      }
    }

    // Recover current search path for custom class loader.
    ClassLoader cl = Thread.currentThread().getContextClassLoader();

    URL urls[] = ((URLClassLoader) cl).getURLs();
    if (urls.length == 0) {
        urls = new URL[1];
        try {
      urls[0] = new URL(VMWConstants.kaffebug);
        } catch (MalformedURLException e) {
      System.out.println("Malformed filename in VMWConstants(String
          kaffebug)");
      System.exit(-1);
        }
    }
    VMWClassLoader customLoader=new VMWClassLoader(urls);

    try {
      Class mainClass = customLoader.loadClass(args[0]);

      // Check using reflection that main Java
      // app has a function main(String[])
      Class parameters[] = new Class[1];
      parameters[0] = String[].class;

      Method method = mainClass.getMethod("main", parameters);
      Object[] arguments = new Object[1];
      arguments[0]=args2;

      method.invoke(null, arguments);
    } catch (ClassNotFoundException e) {
      // The requested main Java app wasn't found.
      System.out.println("The class "+args[0]+" was not found.");
      System.exit(-1);
    } catch (NoSuchMethodException e) {
      // Reflection didn't find main function.
      System.out.println("It seems "+args[0]+" doesn't have a main.");
      System.exit(-1);
    } catch (Exception e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
}
```

9.1.6 VMWConstants.java

Listing 4: VMWConstants.java

```
public class VMWConstants {
    /* path to config file */
    public static final String configFile = "/etc/VMWcfg.xml";
    /* name of VMW library file */
    /* on linux the name will be automatically converted to "lib"+
        libFile+".so"
       and on windows the name will be libFile+".dll" */

    public static final String libFile = "JavaVMW";

    /* solve a little bug in Kaffe */
            public static final String kaffebug = "file:///";
    //    public static final String kaffebug = "file:///home/menghraj
        /undergrad/epfl_4/sem_proj_epxa/vmw/";
    public static final boolean debug = true;
}
```

9.1.7 ClassIdMethodSignature.java

Listing 5: ClassIdMethodSignature.java

```
/*
 * Created on Jan 15, 2004
 *
 * Class to encapusulate a String (representing a class) and
 * a LinkedList of IdMethodSignatures (which itself is a Class to
 * encapsulate two Strings and an int).
 *
 * I have just proven how intrusive Java can sometimes become.
 *
 */

/**
 * @author Alok Menghrajani
 *
 */

import java.util.*;

public class ClassIdMethodSignature {
  public String className;
  public LinkedList idMethodSignature;

  public ClassIdMethodSignature(String c) {
    this.className = c;
    this.idMethodSignature = new LinkedList();
  }
}
```

9.1.8 IdMethodSignature.java

Listing 6: IdMethodSignature.java

```
/*
 * Created on Jan 15, 2004
 *
 * Class to encapusulate two Strings (the first representing a method,
 * the second representing a signature) and an int (representing the
     id in the code).
 * This int is used for optimization (don't need to pass Strings to
     the native library).
 *
 * I wish there was a way to avoid the creation of such silly classes
     ...
 *
 */

/**
 * @author Alok Menghrajani
 *
 */

public class IdMethodSignature {

    public String method;
    public String signature;
    public short id;
    public boolean flag;
    /* This flag is used to display a warning if
     * a method is specified in the XML config file but not found
     * in the actual class.
     */

    public IdMethodSignature(short id, String m, String s) {
  this.method = m;
  this.signature = s;
  this.id = id;
  this.flag = false;
    }
}
```

9.1.9 VMWSkel.java

Listing 7: VMWSkel.java

```
/*
 * Created on Jan 15, 2004
 *
 * Skeleton Class to build VMW.class.
 * I could do things without this class, but it's easier this
 * way (and it is more flexible since changing this class is easy).
 *
 */

/**
 * @author Alok Menghrajani
 */

public class VMWSkel {
    static {
  System.loadLibrary(VMWConstants.libFile);
    }
}
```

9.1.10 VMWcfg.xml

Listing 8: VMWcfg.xml

```
<?xml version="1.0"?>
<methodList>
  <method id="1" class="AnotherClass" name="patchHim" signature="([S[S
      [S)V">
  <map>test.sni</map>
  <methods maxid="4">
    <lookupInfo id="1">
      <kind>instance</kind>
      <class>java/lang/String</class>
      <name>&lt;init></name>
      <signature>()V</signature>
    </lookupInfo>
    <lookupInfo id="2">
      <kind>static</kind>
      <class>java/lang/System</class>
      <name>gc</name>
      <signature>()V</signature>
    </lookupInfo>
    <lookupInfo id="3">
      <kind>instance</kind>
      <class>java/io/PrintStream</class>
      <name>println</name>
      <signature>()V</signature>
    </lookupInfo>
    <lookupInfo id="4">
      <kind>instance</kind>
      <class>java/io/PrintStream</class>
      <name>println</name>
      <signature>(Ljava/lang/String;)V</signature>
    </lookupInfo>
  </methods>
  <fields maxid="1">
    <lookupInfo id="1">
      <kind>static</kind>
      <class>java/lang/System</class>
      <name>out</name>
      <signature>Ljava/io/PrintStream;</signature>
    </lookupInfo>
  </fields>
</method>

</methodList>
```

9.1.11 VMWXMLcfg.dtd

Listing 9: VMWcfg.dtd

```
/*
 * Created on Jan 14, 2004
 *
 */

/**
 * @author Alok Menghrajani
 *
 */

<!ELEMENT methodList (method*)>
```

```
<!ELEMENT method (map, methods*, fields*)>
<!ELEMENT map (#PCDATA)>
<!ELEMENT methods (lookupInfo+)>
<!ELEMENT fields (lookupInfo+)>
<!ELEMENT lookupInfo (kind, class, name, signature)>
<!ELEMENT kind (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT signature (#PCDATA)>

<!ATTLIST method
        id CDATA #REQUIRED
  class CDATA #REQUIRED
  name CDATA #REQUIRED
  signature CDATA #REQUIRED>
<!ATTLIST methods maxid CDATA #REQUIRED>
<!ATTLIST fields maxid CDATA #REQUIRED>
<!ATTLIST lookupInfo id CDATA #REQUIRED>
```

9.1.12 IdeaPlus.java

Listing 10: IdeaPlus.java

```
public class IdeaPlus {

    /* Number of IDEA rounds  */
    private static final short IDEA_ROUNDS = 8;

    /* Number of IDEA subkeys */
    public static final short IDEA_SK_NUM = (6 * IDEA_ROUNDS + 4);

    /* low significant 16-bit */
    private static short lsw16(int y){
  return (short)(y & 0xffff);
    }

    /* most significant 16-bit */
    private static short msw16(int y){
  return (short)((y>>16) & 0xffff);
    }

    /* 2**16 + 1 */
    private static final int MULMOD = (1<<16) | 1;

    // Multiplication modulo 2**16 + 1
    private static short mul(short x, short y)
    {
  short t16;
  int t32;

  x = lsw16(x-1);
  t16 = lsw16(y - 1);
  t32 = (int) ((char)x * (char)t16 + (char)x + (char)t16 + 1);
  x = lsw16(t32);
  t16 = msw16(t32);
  x = (short)(((char)x - (char)t16) + (((char)x)<=((char)t16) ? 1 : 0)
      );
  return x;
    }
  // Compute multiplicative inverse of x by Euclid's GCD algorithm.
  private static short mul_inv(short x)
  {
int n1 = MUL_MOD;
int n2 = (int)(char)x;
int b1 = 0;
int b2 = 1;
int q, r, t;

if ((char)x<=1)
    return x;

while (true) {
    r = n1 % n2;
    q = n1 / n2;
    if (r==0) {
  if (b2<0)
      b2 += MUL_MOD;
  return lsw16(b2);
    }
    else {
  n1 = n2;
  n2 = r;
  t = b2;
  b2 = b1-(q*b2);
  b1 = t;
    }
}
  }


  // Computes IDEA encryption subkeys.
  public static void idea_encrypt_subkeys
(short[] key, short[] subkeys)
  {
int i;

for (i=0;i<key.length;i++)
    subkeys[i] = key[i];

for (;i<IDEA_SK_NUM;i++)
    subkeys[i] = lsw16(((char)(subkeys[((i+1) % 0x8)!=0 ? i-7 : i
        -15]) << 9) |
           ((char)(subkeys[((i+2) % 0x8) < 2 ? i-14 : i-6]) >> 7));
  }

  // Computes IDEA decryption subkeys from encryption subkeys.
  public static void idea_decrypt_subkeys
(short[] encrypt_subkeys, short[] decrypt_subkeys)
  {
int pen = 0;
int pde = 0;
short[] t = new short[encrypt_subkeys.length];
int pt = 0;
int i;

t[6 * IDEA_ROUNDS] = mul_inv(encrypt_subkeys[pen++]);
t[6 * IDEA_ROUNDS + 1] = lsw16(-encrypt_subkeys[pen++]);
t[6 * IDEA_ROUNDS + 2] = lsw16(-encrypt_subkeys[pen++]);
t[6 * IDEA_ROUNDS + 3] = mul_inv(encrypt_subkeys[pen++]);

for (i=6*(IDEA_ROUNDS-1);i>=0;i-=6) {
        t[i + 4] = encrypt_subkeys[pen++];
        t[i + 5] = encrypt_subkeys[pen++];
        t[i] = mul_inv(encrypt_subkeys[pen++]);
        if (i!=0) {
      t[i + 2] = lsw16(-encrypt_subkeys[pen++]);
      t[i + 1] = lsw16(-encrypt_subkeys[pen++]);
        }
        else {
      t[1] = lsw16(-encrypt_subkeys[pen++]);
      t[2] = lsw16(-encrypt_subkeys[pen++]);
        }
        t[i + 3] = mul_inv(encrypt_subkeys[pen++]);
    }

    for (i=0;i<IDEA_SK_NUM;i++) {
        decrypt_subkeys[pde++] = t[pt];
        t[pt++] = 0;
    }
      }


      //IDEA encryption/decryption algorithm.
      // Note: block_in and block_out can be the same block.
      public static void idea_cipher
    (short[] block_in, short[] block_out, short[] key)
      {
    int pin = 0;
    int pout = 0;
    int pk = 0;
    short word1, word2, word3, word4;
    short t1, t2;
    int i;

    word1 = block_in[pin++];
    word2 = block_in[pin++];
    word3 = block_in[pin++];
    word4 = block_in[pin++];

    for (i=IDEA_ROUNDS;i>0;i--) {
        word1 = mul(word1,key[pk++]);
        word2 = lsw16((char)word2 + (char)key[pk++]);
        word3 = lsw16((char)word3 + (char)key[pk++]);
        word4 = mul(word4,key[pk++]);

        t2 = (short)(word1 ^ word3);
        t2 = mul(t2,key[pk++]);
        t1 = lsw16((char)t2 + (char)(word2 ^ word4));
        t1 = mul(t1,key[pk++]);
        t2 = lsw16((char)t1 + (char)t2);

        word1 ^= t1;
        word4 ^= t2;

        t2 ^= word2;
        word2 = (short)(word3 ^ t1);
        word3 = t2;
        //System.out.println("Round "+i+" :"+Integer.toHexString((char)
            word1)+" "+Integer.toHexString((char)word2)+" "+Integer.
            toHexString((char)word3)+" "+Integer.toHexString((char)word4
            ));
    }
```

```
  word1 = mul(word1,key[pk++]);
  block_out[pout++] = word1;
  block_out[pout++] = lsw16((char)word3 + (char)key[pk++]);
  block_out[pout++] = lsw16((char)word2 + (char)key[pk++]);
  word4 = mul(word4,key[pk]);
  block_out[pout] = word4;
  //System.out.println("Round 0:"+Integer.toHexString((char)word1)
      +" "+Integer.toHexString((char)word2)+" "+Integer.toHexString((
      char)word3)+" "+Integer.toHexString((char)word4));
    }

}
```

9.1.13 IdeaTestSw.java

Listing 11: IdeaTestSw.java

```
public class IdeaTestSw {

    public static void main(String[] args)
    {

  if (args.length < 1) {
      System.out.println("Usage: java IdeaTestSw <integer>");
      return;
  }

  int i,j;
  int fd;
  int data_in;
  int data_out;

  int[] param = new int[4];
  int[] obj0 = new int[2];
  int n, n64;

  try {
      n64 = (new Integer(args[0])).intValue();
  }
  catch(NumberFormatException nfs) {
      System.out.println("You must provide an integer");
      return;
  }
  n = n64*2;

  short[][] key = new short[3][8];
  short[] enkey = new short[IdeaPlus.IDEA_SK_NUM];
  short[] dekey = new short[IdeaPlus.IDEA_SK_NUM];
  short[][] rdata = new short[n64][4];
  short[][] rdata_out = new short[n64][4];

  /*
    ==========================================================
                         IDEA reference code
    ==========================================================
  */

  for (i=0; i<8; i++)
      key[0][i] = (short)(i+1);
```

```
for (i=0; i<8; i++)
    System.out.print((Integer.toHexString(key[0][i]))+"  ");

/* Test with only one key */
IdeaPlus.idea_encrypt_subkeys(key[0],enkey);
System.out.println();
for (i=0; i<8; i++)
    System.out.print((Integer.toHexString((char)enkey[i]))+"  ");
IdeaPlus.idea_decrypt_subkeys(enkey,dekey);
System.out.println();
for (i=0; i<8; i++)
    System.out.print((Integer.toHexString((char)dekey[i]))+"  ");

System.out.println();
System.out.println("Input blocks:");
for (j = 0; j < n64; j++) {
    for (i=0 ; i<4; i++) {
  rdata[j][i]= 0;//(short)(j*10 + i);
  System.out.print((Integer.toHexString((char)rdata[j][i]))+"\t");
      }
    System.out.println();
}

/* Encryption starts */
/*if ((fd = open("/dev/fpga",O_RDONLY)) == -1 )
    {
  printf("Can't open fpga");
  exit(1);
    }*/

/*if (ioctl(fd,FPGA_TIMER_INIT,1) != 1)
    {
  printf("Error initalizing timer...\n");
  return 1;
    }*/

for (j = 0; j < n64; j++) {
    IdeaPlus.idea_cipher(rdata[j], rdata_out[j], enkey);
}
System.out.println();
System.out.println("Output blocks:");
for (j = 0; j < n64; j++) {
    for (i=0 ; i<4; i++)
  System.out.print((Integer.toHexString((char)rdata_out[j][i]))+"\t"
      );
    System.out.println();
}

/*if (ioctl(fd,FPGA_TIMER_STOP,1) != 1)
  {
  printf("Error stoping timer...\n");
  return 1;
  }*/
System.out.println("This is the SW time measured for encryption of "
    +n64+" 64-bit blocks");

/* Encryption ends */

/*
   ==================================================================
                          IDEA reference code END
   ==================================================================
  */

    }

}
```

9.2 Bibliography

References

[1] Joshua Engel, *Programming for the Java Virtual Machine*, Addison Wesley, second edition, 1999.

[2] Java API, *http://java.sun.com/j2se/1.4.2/docs/api/*

[3] BCEL API, *http://jakarta.apache.org/bcel/apidocs/index.html*

[4] BCEL browsable source, *http://jakarta.apache.org/bcel/xref/index.html*

[5] Miljan Vultic, Ludovic Righetti, Laura Pozzi, Paolo Ienne, *Portable Reconfigurable Coprocessors through Interface Virtualization*, EPFL.

[6] Ludovic Righetti, Miljan Vuletic, *Operating System Support for Reconfigurable System on Chip*, EPFL semester project, 2003.

[7] Christophe Dubach, Miljan Vuletic, *Java Virtual Machine on FPGA based platforms*, 2004.