A Lightweight Java Virtual Machine for a Stack-Based Microprocessor

Mirko Raner
PTSC
(formerly "Patriot Scientific Corporation")
10989 Via Frontera
San Diego, CA 92127
raner@acm.org

Abstract

The large majority of modern JVM implementations are either pure software VMs on top of standard general purpose microprocessors (e.g., Insignia's Jeode or IBM's Jalapeño VM) or Java-specific microprocessors with supportive software layers (e.g., Fujitsu's MB 86799 or aJile's aJ-100).
In this paper a somewhat different approach is presented: a lightweight software VM on top of a general purpose *stack-based* microprocessor. Said microprocessor is not a "hardware JVM", but its architecture is very similar to the JVM. Being truly a general purpose processor, it is not entirely dedicated to execute Java bytecode but can at the same time run C or FORTH applications.

This paper describes the implementation of a lightweight Java Virtual Machine for the IGNITE family of stack-based microprocessors. To achieve an optimal Java performance this JVM uses a combination of standard techniques, such as ahead-of-time (AOT) compilation, class hierarchy analysis (CHA), lazy class loading and binary rewriting, complemented by new optimizations like executable method access structures (XMAS) and lazy argument passing.
The unusual architecture of the target processor also often posed unusual problems that had to be solved.

1 Introduction and Project Background

Originally, the IGNITE microprocessor (formerly named PSC 1000A [PTS00]) was designed as an embedded computing platform for efficient execution of C and FORTH. It has a 32-bit dual-stack architecture (see figure 1) with an 18-word operand stack and a separate 16-word stack for the call stack frames (containing the local variables). Both stacks act as on-chip stack caches and are automatically spilled to and refilled from memory.
The processor uses byte-sized instructions, fitting up to four instructions into one 32-bit machine word. Thus, the IGNITE processor can be considered a real "bytecode" processor (though not a *Java* bytecode processor). Special instruction formats are used for encoding branches and 8-bit or 32-bit constants.
Simplicity was the main design goal of the processor. To save space on the silicon die, it does not have conventional data or instruction caches, and instead of a multi-stage pipeline architecture it uses a simple instruction pre-fetch.

When Java became a popular programming environment, the obvious similarities between the IGNITE microprocessor and the JVM inspired a port of a PersonalJava Application Environment (PJAE). This port was based on Wind River Systems' VxWorks 5.4 as underlying real-time operating system and the Personal JWorks 3.0.2 PJAE ("PJWorks") [Win99]. To take advantage of the IGNITE's architectural similarities to the JVM, a JIT compiler was added to PJWorks and the interpreter loop was recoded in assembly language.

Surprisingly at first, the actual performance of this JVM port fell far behind the calculated estimates. The scores of the CaffeineMark test reached on average only about 40% of the expected scores – even with JIT compilation and a hand-coded interpreter loop! Though the JIT compiler yielded performance gains of up to 10 times over the original interpreter loop, the system was still about 2.5 times slower

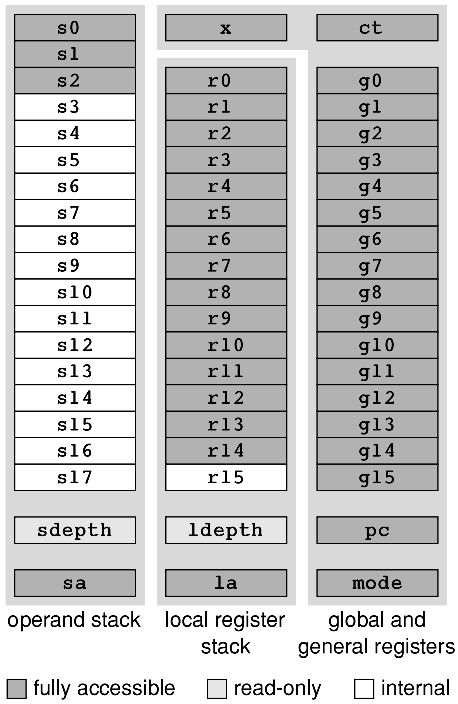

Figure 1: The register set of the IGNITE I

than predicted. A closer examination revealed that the PJWorks design (based on Sun's 1998 version of the JVM) caused a number of severe overheads for the IGNITE processor. Those problems could hardly be overcome without changing the overall architecture completely.
Eventually, the insufficiencies of the PJAE port led to the idea of a new optimized JVM for the IGNITE platform.

Section 2 gives an overview of the new Java architecture for the IGNITE processor; it presents an analysis of the problems of the earlier PJAE port (2.1) and summarizes the design goals of the new system (2.2). Section 3 introduces the low-level OS API used for the new JVM and section 4 covers the implementation details of the new lightweight virtual machine itself. The optimized method invocation scheme of the VM is covered separately in section 5. Finally, section 6 provides preliminary benchmark results and concludes the paper.

2 Overview of the New Architecture

The outcome of the redesign effort, after the disappointing results with the initial PJAE port, was the Lightweight Virtual Machine (LVM) with its low-level API named "JELLO".
For reasons of simplicity and rapid prototype development, the first version of the LVM is based on the reduced instruction set of the CLDC specification – even though the IGNITE processor in fact *does* have floating point instructions.

2.1 Problem Analysis for the Initial PJAE Port

It turned out that the reasons for the low performance of the initial PJAE port were distributed across all layers of the system: the VM implementation, the underlying OS and the microprocessor itself.
The deficiencies were not only located within the components themselves: a lot of the problems were actually caused by the interaction *between* the individual components.

One of the main reasons for the performance limitations was found in the predefined architecture and data structures. Sun's PJAE reference implementation, which is also the core of Wind River's Personal JWorks 3.0.2, was designed for high portability and not necessarily for high performance. It contains a number of abstraction layers which facilitate porting to new platforms but often impede an efficient implementation on a particular processor.
Though a lot of the crucial JVM code was recoded in assembly language there were still large overheads imposed by the interaction with Sun's and Wind River's parts of the VM. Especially jumping between different types of methods (interpreted, JIT-translated, JNI-native, NMI-native, etc.) turned out to be very expensive in some cases (see section 5.4 for more information about "invokers"). In fact, the classical JIT compiler approach is completely inadequate for a target architecture that is so similar to the JVM.

Being originally targeted for the C and FORTH programming languages, the IGNITE processor also uses a different "procedure call standard" than the JVM. That is, it uses a different way of passing arguments to a called procedure (or method). Furthermore, the IGNITE processor uses a different notion of the concept of "stack frames": the call stack (the "local register stack" in IGNITE nomenclature) and the operand stack are separate structures, and the procedure call logic has to establish links between the call stack and the corresponding entries in the operand stack.
Those problems are processor-specific problems.

However, not being limited to the predefined mechanisms and data structures of the reference implementation makes it much easier to find optimized solutions for these problems.

The original PersonalJava VM was mainly written in C and naturally made frequent use of C `structs` to group and handle data. The C language allows structures to be passed as arguments, but the IGNITE architecture is not quite made for that, because the on-chip stack caches have a very limited size and procedure calls get very inefficient when the arguments do not fit into the cache. Therefore it is necessary to use an auxiliary stack, which solves the problem but introduces another considerable overhead.

Finally, VxWorks was also not the best choice for the base operating system. The thread models of Java and VxWorks are fundamentally different and it takes considerable efforts to make one run on the other. In addition to that, the real-time requirements of VxWorks and the non-real-time nature of Java are hard to reconcile, since real-time behavior is hard to guarantee in an environment that depends on automatic garbage collection (though there are approaches that enable real-time garbage collection).
Personal JWorks is also very closely tied to VxWorks, which was another problem, since it made it very hard to change to a different operating system.

Besides the problems of lacking performance and flexibility, the port of Personal JWorks also had an excessively large memory footprint. After JVM-startup about 5 MB of memory had already been consumed, which is a very large amount for a Java solution that is intended to be used in embedded applications.

2.2 Design Goals for the New Java Architecture

After the detailed problem analysis of the earlier JVM port, the following design goals were established for the optimized IGNITE JVM architecture:

1. Increase the Java performance
2. Decrease the memory footprint size of the overall OS/VM combination
3. Take advantage of the unique processor features of the IGNITE
4. Provide the flexibility to use different underlying operating systems
5. Avoid the problems that were caused by choosing C as implementation language

The footprint goal was to have a combined static/-dynamic footprint of about 2 MB for the JVM, including low-level OS functionality required by the JVM.

2.3 Feature Overview

Various different measures were taken in order to meet all the goals stated in section 2.2.
In comparison to the previous Virtual Machine port, the new architecture has four major optimizations:

- Proper separation from the underlying OS
- Lazy class resolution
- Pure Ahead-of-Time compilation without JIT or interpreter
- Optimized method invocation

Probably the most interesting is the optimized method invocation, which is discussed separately in section 5. The optimization features were especially aimed at design goals 1, 2 and 3.
In order to gain more flexibility towards the choice of the underlying operating system (design goal 4), an OS abstraction layer – called "JELLO" – was introduced. JELLO is covered in more detail in section 3.

Finally, in order overcome the implementation problems with C (auxiliary stack for `structs`, different procedure call standards), the LVM was entirely implemented in Java and assembly language. The Java parts (such as the AOT compiler) are converted to IGNITE code via bootstrapping. Using Java as main implementation language has already proven to be a successful idea in other JVM projects, such as the Jalapeño VM [A+00]. The current implementation of the low-level API is still written in C and is based on PTSC's "monitor", which is a minimal operating system for the IGNITE processor.
All JELLO interface functions were specified on machine level (by specifying the operand stack contents before and after a low-level system call).

3 The Low-Level API (JELLO)

The JELLO API (Java Environment Low-Level Operations API) is an API that facilitates the implementation of the Java Virtual Machine on top of an

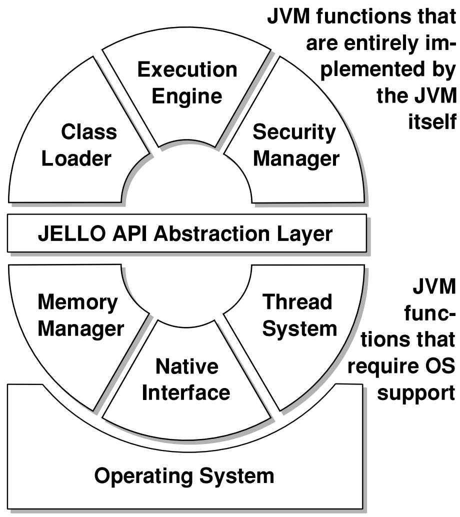

Figure 2: The JELLO API Layer

underlying operating system that is running on the IGNITE processor architecture.
JELLO is designed for the IGNITE processor but it can also be used in addition to other OS/JVM abstraction layers, if the design of the particular API does make sense in conjunction with JELLO.

3.1 API Design

The design of the JELLO API is based on an analysis of the interaction between the JVM and the OS in the classic JVM model [Ran99]. Whereas execution engine, class loader and security manager can be independently implemented in the JVM, the remaining components – thread scheduler, memory manager and native interface – are directly dependent on OS functionality for their implementation.
Of course, *indirectly* all components depend on OS functionality. However, the functionality can be accessed through lower-level components of the JVM itself. For instance, the class loader will need to allocate memory for new classes and perform file I/O, but in order to do so, it can use the memory manager and native interface of the JVM (through JELLO) – no *direct* access of OS functions is necessary (see figure 2).

Especially when it comes to memory management and thread scheduling, the JVM has very specific demands. Thread prioritizing and context switching has to be handled in a certain way, and memory management has to support automatic garbage collection. However, most operating systems do not support garbage collection and their multi-threading API significantly differs from the Java thread model.
The JELLO API encapsulates all the OS functionality that is needed by a JVM and wraps all functions in a way so that they can be directly used for the JVM implementation. For example, the `/setPriority` function of JELLO uses the same priority levels as the `setPriority` method of Java's `java/lang/Thread` class and the memory allocation functions allow passing of additional type information for the garbage collector.
This orientation towards the JVM – as opposed to an orientation towards the OS – distinguishes JELLO from other low-level implementation APIs (such as Sun's Java HPI (Host Programming Interface); [LY97]). In order to change to a different operating system, only the JELLO API layer has to be ported, the JVM on top of JELLO does not require any modifications.

JELLO does not only separate the JVM from the OS but it also isolates the JVM as far as possible from language-, processor- or OS-related idiosyncrasies (such as the auxiliary C stack or a Java-incompatible OS thread model).
However, the JELLO API is specifically for the IGNITE processor and will not make much sense for other processor architectures.

3.2 JELLO Object Layout

JELLO does not know anything about the internal data structures of the JVM. The JELLO API specifies only a small number of data structures that are necessary for the communication between JELLO and the JVM. The most essential data structure specified by JELLO is the layout of JELLO objects in memory (see figure 3; the diagram uses the OVAL [Ran00] notation).
Because JELLO is responsible for memory allocation and automatic garbage collection, the JVM at least needs to convey the size of an object and pointer map information for the garbage collector.

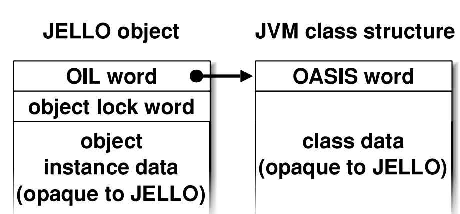

Figure 3: JELLO Object Layout

31	6	1 0
pointer map	size	0 0
pointer to structure		1 0
size of primitive array		0 1
size of object array		1 1

Table 1: OASIS words for small objects, large objects, primitive arrays and object arrays

This is done by the "OASIS" word – containing the Object Allocation and Storage Information Structure.

Every JELLO object has a 2-word header: the first word is always a pointer (!) to the OASIS word; this pointer is also called the Object Identification Link (OIL). The second word is reserved for the object lock. Currently the second word is just a plain pointer to a monitor structure but more clever mechanisms like inflatable locks or relaxed locks [Dic01] will be used in future versions of the LVM.

From the third word of the object onwards, all data is private data, which is opaque to JELLO. The garbage collector knows how big the object is in memory and which of the words after the 2-word header are pointers. All other details of the object layout are entirely up to the JVM implementation.

The OASIS word will usually be the first word of a larger object identification structure. This structure can be the `class_info` or class block – but that does not necessarily have to be so. In fact, the LVM uses this space for the virtual method table (VMT). Slot 0 is reserved for the OASIS word, slot 1 is always used for the `getClass()` method, which gives access to the full class data.

Two objects that have the same OIL pointer value (that is, they point to the same OASIS word in memory) always belong to the same class.

Depending on the class or array that it describes, the OASIS word can have different forms (table 1). For small objects, all required information is stored in the OASIS word itself, for larger and more complex objects the OASIS word turns itself into a pointer. The nonpointer form can be used for all objects that do not have more than 128 bytes of instance data and contain references only in the first 25 words (100 bytes) of the object. These restrictions apply because the OASIS word for small objects uses 5 bit for storing the object size ($4 * 2^5 = 128$) and 25 bit for the pointer map.

Objects that do not qualify as "small", use the pointer form of the OASIS word, which points to one word that contains the object size, followed by one or more pointer map words (one of the 32 bits is used as an "end" marker).

JELLO offers an API for allocating memory (for use by `new`, `newarray` etc.) and explicit GC. It also guarantees that automatic GC is performed in a JVM-compliant manner. However, it does not provide an API specifically for the implementation of GC algorithms (like, e.g., that of the EVM [WG98]). A GC implementation for the LVM must adhere to the JELLO memory layout and provide adequate implementations for the GC-related JELLO functions. It can use any functionality supplied by the OS or by other JELLO components.

4 LVM – The Lightweight Virtual Machine

The main improvements of the LVM itself were already listed in section 2.3: AOT compilation, "lazy" class loading and an optimized method invocation scheme. AOT compilation and lazy class loading are discussed in sections 4.1 and 4.2. Method invocation is comprehensively covered in section 5. Sections 4.3 and 4.4 deal with issues of garbage collection and the native interface of the LVM.

4.1 Ahead-of-Time Compilation

The goal of traditional "just in time" compilation is to find a reasonable compromise between the performance gained by executing compiled methods and the performance lost during the compilation process itself. Usually, only selected methods will be compiled, while others remain interpreted in order to avoid the time penalty for their translation.

This is a sensible approach for most microprocessor architectures, since often very complicated register mapping problems have to be solved in order to compile Java bytecode for a non-stack-based microprocessor.

One disadvantage of JIT compilation is that Java class files have to be kept in memory. Freeing memory used for the bytecode of individual methods within a class file is possible, but is very complicated and can lead to memory fragmentation so that a lot of JIT compilers don't actually do it. Another problem of the JIT approach is the potentially very costly interaction between JIT-compiled and interpreted code (see also sections 2.1, 5.4).

The LVM completely translates a class file immediately after it has been loaded. All information from the class file is retained in the translated version of

the class and the original class file is discarded. All class files are loaded into a single designated temporary buffer.
Translation delays are not an issue for the LVM because most Java bytecode instructions can be mapped one-to-one to corresponding IGNITE instructions or short instruction sequences; the translation process is basically a simple table lookup.

The ahead-of-time (AOT) approach was primarily chosen to avoid the previously mentioned disadvantages of JIT compilation. Also, the compromise between translation time penalties and performance gains is not necessary on the IGNITE platform, as the translation time is negligible in comparison to the loading time for a class file.

4.2 Lazy Class Loading

Class loading and resolution can be performed in a "static" or in a "lazy" manner [LY99, §2.17.3]. With a few limitations, a JVM implementation can freely decide when a particular class is loaded and resolved.
For example, a simple test can be used to reveal the class loading policy of Sun's Java 2 VM. On a UNIX system, using the command line

```
java -verbose:class HelloWorld|grep Loaded|wc -l
```

one can determine how many classes are loaded for a simple `HelloWorld` program.*) A surprising class count between 160 and 210 can be observed for most Sun Java 2 VMs – for a simple `HelloWorld`.
In some JVM implementations loading of one particular class can spawn a whole tree of other classes that are loaded.

The LVM defers the resolution of classes and loading of additional classes as much as possible. The minimum requirement is that the superclass of every newly loaded class must be loaded and resolved before that class. This is necessary in order to properly construct the virtual method tables and the memory layout of objects. Besides that, the LVM always leaves external references unresolved until the moment they are actually used for the first time. So, for example, instead of loading an exception class just because some method declares that it might throw that type of exception at some point in time, the loading of the exception class is delayed until that particular exception is actually thrown (or, more precise, until an exception object is created).

The method invocation scheme of the LVM allows that references to classes, methods and fields can stay unresolved until they are accessed; for details see section 5.

*) an additional `2>&1` might be necessary for some JVMs that print verbose messages on `stderr` instead of `stdout`.

4.3 Garbage Collection Algorithms

Currently, the LVM uses a simple, merely conservative mark & sweep garbage collector. The JELLO object memory contains accurate pointer maps for all objects, however, the stack frames on the call stack do not have pointer maps yet. Later versions of the LVM might provide pointer maps for the stack frames as well.
With the exception of hybrid garbage collectors that also do reference counting, most common garbage collection algorithms can be used with the LVM. Reference counting is problematic because additional reference adjusting code would need to be added and the one-to-one translation between Java bytecode and IGNITE code would be lost. For example, `dup` can be translated to the IGNITE instruction `push s0`; if reference counters are involved, this simple translation will no longer work, because the reference count of the top-of-stack object needs to be adjusted as well.

Section 3.2 contains further details on memory layout and garbage collection in the LVM.

4.4 Native Methods

The LVM is modeled after the CLDC specification and therefore it does not need to implement the Java Native Interface (JNI).
Still the LVM has to provide a way to implement methods that cannot be written in Java because they need to access low-level OS functionality.
In addition to this basic native method problem, there is also the question how the LVM classes themselves can access low-level functionality such as reading and writing memory. The Jalapeño VM, for example, solves the latter problem by its MAGIC class [A+00, appendix A].

In the LVM both problems are solved with the same technique. Similar to the `asm` directive of many C and C++ compilers, the LVM offers an IGNITE inline assembly directive. Table 2 shows an example of IGNITE native programming.†) Java source code

†) In fact, the listing does not show the actual implementation of the hashcode function; the shown implementation was chosen to demonstrate the use of labels and branches in inlined native code; the real implementation simply uses 3 `SHR_1` instructions instead of a complicated loop.

```
package com.ptsc.lvm.java.lang;
import com.ptsc.lvm.java.Native;
public class System implements Native
{
    /** Returns object address >> 3. **/
    public static int
    identityHashCode(Object object)
    {
        int[] $native =
        {
            PUSH_S0,       // clear carry
            AND,
            PUSHN_3,       // do 3 shifts
            POP_CT,        // set loop ct
        ~1, SHR_1,         // shift
            DBR, ~1,       // loop back
            RET
        };
        return $INT;
    }

    /** Explicitly trigger a GC. **/
    public static void gc()
    {
        int[] $native =
        {
            BR, $._("/gc")// call JELLO
        };
        return;
    }
```

...

Table 2: An example for LVM native programming

that contains inlined assembly code can still be compiled with any standard Java compiler (though it looks very unusual at first). When the class files for such native classes are executed on a standard JVM, they will produce an error – or simply do nothing in the best case. However, the AOT compiler of the LVM will recognize the special content of those classes and produce the equivalent IGNITE machine code. In fact this technique is pretty common and is used in Jalapeño as well as in Jbed [TMH99].

The `Native` interface contains integer constants for all IGNITE instructions. The spelling of the instruction mnemonics has been slightly adapted, so that the constant identifiers conform to the Java syntax and the Java naming conventions [GJSB00, §6.8.5]. For instance, `push s0` turns into `PUSH_S0` and `br []` becomes `BR_$$`.
Besides the instruction mnemonics, the `Native` interface also provides constants that indicate the return type of a method (e.g., `$INT` or `$BYTE`), a general label symbol (`LABEL-` or `~`) and a symbolic reference operator (`$`).
Labels are defined and referred as `~0`, `~1`, etc. or `LABEL-1`, `LABEL-2` etc. JELLO API functions or symbolic method references can be located with `$._("`⟨*method name*⟩`")`.

The class names of the JDK library classes for the LVM are prepended with `com.ptsc.lvm` so that they can be compiled with standard Java compilers. Again, the LVM AOT compiler will convert the package specifications back to the original JDK package names.

5 Optimized Method Invocation

Method invocation is one of the most crucial operations in an object-oriented system. If method invocation is slow or subject to unnecessary overheads, this will also degrade the overall system performance.
For this reason, method invocation in the LVM was optimized with the following techniques:

- Devirtualization based on Class Hierarchy Analysis (CHA)
- Executable Method Access Structures (XMAS)
- Binary rewriting
- Lazy argument passing

Those optimizations, and how they apply for the LVM, are described in more detail in sections 5.3 to 5.8. Section 5.1 describes in general how Java-style method invocation can be implemented on the IGNITE platform and section 5.2 deals with IGNITE-specific stack frame problems.

5.1 Phases of Method Invocation

Procedure calls on the IGNITE dual-stack architecture and method invocations in the JVM are substantially different.
The Java Virtual Machine passes arguments on the operand stack, but for the called method the arguments appear in the local variables (which correspond to the IGNITE's local register stack). On the IGNITE the arguments stay on the operand stack and need to be moved to the right place so that the computation model is not violated.
The `invoke`... instructions of the JVM also perform a lot of additional functions, besides merely transferring execution to another place. They take care

	Operation	Comments
1	Resolve method	needs only be done once for previously unresolved methods
2	Check whether the object is `null`	not necessary for `static` methods; can be omitted if object is "`this`"
3	Find correct method code	only necessary for true virtual methods; can be optimized in many cases
4	Transfer arguments to local register stack	can be optimized with lazy argument passing
5	Transfer execution to target method	
6	Create register stack space for variables	
7	Establish operand stack link for stack frame	

Table 3: Phases of method invocation

of method resolution, check whether the caller is using a `null` reference, automatically dispatch virtual methods and allocate a new stack frame for the called method. All these things have to be done "manually" on the IGNITE processor.
Table 3 summarizes the phases of method invocation.

5.2 Linking the Call Stack to the Operand Stack

In the computation model of the JVM every stack frame on the caller stack actually *contains* the operand stack for the method. Since the required size of the operand stack is a known constant [LY99, §4.7.3] this does not cause any problem. Stack frames can be implemented such that the operand stack part of one stack frame overlaps with the local variables of the next frame. Overlapping stack frames completely eliminate the need for copying arguments to the local variables of the receiving method and can be implemented very efficiently on a large number of architectures (including SPARC).
Unfortunately, this is not true for the dual-stack architecture of the IGNITE processor.

Operand stack (`s0` – `s17`) and call stack (`r0` – `r15`)

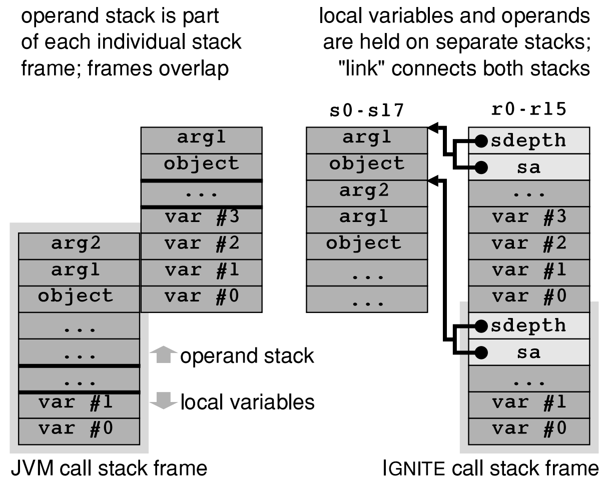

Figure 4: JVM stack frames versus IGNITE stack frames (simplified)

are separate structures in the IGNITE architecture. The operand stack is used for all computations, the call stack only stores variables and the return address.
There are two potential ways to simulate a JVM-style stack structure on the IGNITE:

- Save the whole contents of the operand stack into the local register stack (not only the arguments but also the data that might be on the stack *below* the arguments)
- Leave the residual contents of the operand stack where they are and establish a link between the stack frame on the local register stack and the "operand stack frame" on the operand stack; in case of an exception the link allows a rollback to the correct operand stack state

Clearly, the latter solution is the favorable one, because it involves less data shifting between the two stacks.
Unfortunately, it is not possible to find out the projected memory address of an operand stack element directly. The processor only offers the current base stack pointer for the operand stack (`sa`) and the number of elements currently in the on-chip cache (`sdepth`). The projected address of a stack element in memory can be calculated from those two values. To save time, both `sa` and `sdepth` are stored in the local register stack to form the operand stack link. Only in case of a rollback (that is, an exception) the effective address is actually calculated.
Figure 4 visualizes how overlapping Java stack frames are simulated on a dual-stack architecture.

5.3 Devirtualization based on CHA

Virtual method invocations can often be replaced by faster nonvirtual invocations, which do not require a VMT lookup [DGC95].
Except for constructor calls and invocations of `private` or `super` methods, Java treats all non-static method invocations as virtual. With static and dynamic class hierarchy analysis (CHA) virtual invocations can be converted ("devirtualized") to nonvirtual ones in many cases. This technique is already used by several JVM implementations (e.g., the Harissa VM [MMBC97]).

Java bytecode represents a method reference as a pair of a class reference and a method signature (e.g., `java/lang/Math/round(D)L` or `A/one()V`) [LY99, §4.4.2]. During a virtual method invocation the reference resolves to a particular method implementation, which either belongs to the referred class itself or to a subclass of that class. The referred class might have inherited the method implementation from a direct or indirect superclass, but generally an `invokevirtual` instruction cannot access method implementations of superclasses (`invokespecial` needs to be used in those cases). Together with the loading order of classes used by lazy class loading, this fact can be used for implementing a very simple devirtualization scheme.

The lazy class loader always processes superclasses first, because otherwise it could not set up the virtual method table (VMT) for a class.
Therefore, when a loading request for a particular class is issued, it can be safely assumed that no subclasses of that particular class are present in the system yet (otherwise the requested class would already have been loaded as a superclass, and the loading request would not have been issued in the first place). The LVM takes advantage of this and initially marks *all* methods of a newly loaded class as *devirtualized*. Only, when that class is subclassed and methods are overridden later (or even during the same loading request to the class loader) those methods are reverted to their regular virtual state.

Figure 5 shows a simple class hierarchy (`C extends B extends A`): class `A` defines two methods `one()V` and `two()V`, class `B` overrides `two()V` and adds `three()V`, class `C` again overrides `one()V` and `three()V`.

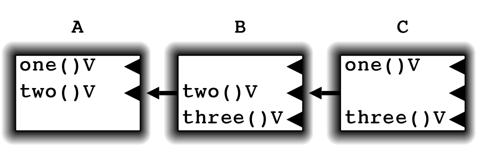

Figure 5: Example class hierarchy (OVAL notation)

class `A` loaded	class `B` loaded	class `C` loaded
♦`A/one()V`	♦`A/one()V`	◊`A/one()V`
♦`A/two()V`	◊`A/two()V`	◊`A/two()V`
	♦`B/one()V`	◊`B/one()V`
	♦`B/two()V`	♦`B/two()V`
	♦`B/three()V`	◊`B/three()V`
		♦`C/one()V`
		♦`C/two()V`
		♦`C/three()V`

♦= devirtualized, ◊= revirtualized

Table 4: Devirtualization of methods

Table 4 shows how the devirtualization status of the methods changes when classes `B` and `C` are added later. It is noteworthy, that, even though some methods are basically the same (e.g., `B/one()V` inherits the code from `A/one()V`), it is necessary to distinguish whether a method invocation refers to `A/one()V` or `B/one()V`.

Sections 5.4 and 5.6 explain in more details how the devirtualization and revirtualization is handled in the LVM.

5.4 Executable Method Access Structures (XMAS)

Every JVM implementation requires data structures that store information about each loaded method. Amongst the stored pieces of information are the address of the method code, the method flags and possibly also the slot offset in the VMT (Virtual Method Table), if such a structure is used. In Sun's JVM terminology those structures are called `method_info` [LY99, §4.6] or "method blocks" [Yel96].
In his document "The JIT Compiler API" Frank Yellin also introduced the concept of "invokers" [Yel96, §6.5]. An invoker is a piece of code that acts as a kind of adapter when execution is transferred between methods of different types. Each method type has a particular invoker associated with it. There are different invokers for interpreted, synchronized interpreted, native, JIT-compiled and other types of methods. Thus, the JVM does not need to know how to call code that has been produced by a particular JIT compiler plug-in; it only has to call the appropriate invoker, which will handle the

	Unresolved	**Devirtualized**	**Revirtualized**
`0x00`	`br 0x08`	`call makeshortcut`	VMT dispatcher code (see table 6)
`0x04`		address of method code	
`0x08`	`push.l` / `push.l`	VMT offset of method	
`0x0C`	pointer to class name	pointer to class structure	pointer to class structure
`0x10`	pointer to signature pair	pointer to signature pair	pointer to signature pair
`0x14`	`br resolve`	access flags	access flags

Table 5: XMAS layout for unresolved, devirtualized and revirtualized methods

interaction.
On the back side, invokers are an additional level of indirection and impose a certain overhead. This overhead adds to any already existing overhead, for example, that of a VMT lookup.

In the LVM the concepts of "method block" and "invoker" are combined into a new data structure which has been given the name "executable method access structure" (nicely abbreviated XMAS).

As the name already suggests, the XMAS contains executable code. Normally, the method dispatching code has to extract the data from the method block in order to transfer execution to the appropriate address.
The LVM uses a slightly different technique: instead of extracting various pieces of information from the data structure and acting accordingly, a method is invoked by simply calling the XMAS like an executable routine. The code in the XMAS *is* already the dispatcher and will transfer execution to the correct address.

Whereas the pointer to the XMAS of a method will always remain the same, the layout and contents of the XMAS will change as the method goes through different stages of its lifecycle. There are different variations of XMAS blocks for unresolved, devirtualized and revirtualized methods. Table 5 shows the different XMAS layouts for methods called by `invokevirtual`.

An unresolved method is a method whose class has not been loaded yet. Since the LVM uses lazy class loading, this situation may appear very frequently. The XMAS for unresolved methods contains a pointer to the class name and a pointer to the signature pair. When the XMAS is executed those two values are pushed onto the stack and the resolution routine (`resolve`) is called. To fully understand the XMAS layout for unresolved methods, one must know that the IGNITE processor allocates a separate word for each `push.l` constant within an instruction group; the constants follow immediately after the instruction group and are automatically skipped by the decoding logic.

Whenever an unresolved method is actually invoked for the first time, the resolver will load the appropriate classes and will change the XMAS to the "devirtualized" state (see also section 5.3).

If a method gets overridden by another method that got newly loaded, the devirtualization will get undone ("revirtualization"), so as to avoid that falsely optimized methods are called.

For revirtualized methods, which consequently require a VMT lookup, the three first words of the XMAS contain the method dispatcher code. Depending on the VMT slot number of the method there are three different formats how those three words are used (see table 6).
As the LVM uses the JELLO Object Identification Link already as the VMT pointer, slot offset 0 will never be used, because slot 0 contains the OASIS word for the class. Slot offset 4 always branches to the `getClass()` method, which gives access to all other details of a loaded class (see also section 3.2). As a further optimization, slot offset 8 is reserved for the `hashCode()I` method.

The LVM uses the VMT data structure in a slightly

push sp	ld []	push.n #i	add
br []			
(#i, repeated)			

(a) Dispatcher code for slot offsets 4 and 8.

push sp	ld []	push.b #i	(#i)
add	br []		
(#i, repeated)			

(b) Dispatcher code for slot offsets 12 to 252.

push sp	ld []	push.l #i	add
(#i)			
br []			

(c) Dispatcher code for slot offsets 256 and higher.

Table 6: (a) – (c) Dispatcher code sequences.

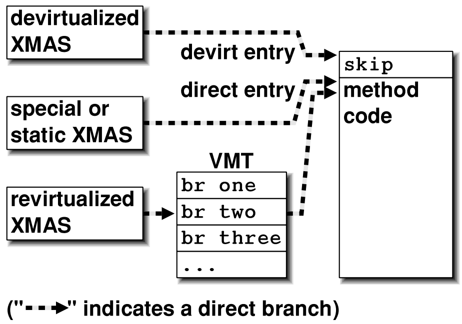

Figure 6: Relations between XMAS and VMT.

different manner compared to other object oriented systems.

Instead of reading the VMT slot contents (instruction `ld []`) and then branching (`br []`), the dispatcher code directly uses `br []`. This is possible because the VMT does not contain the address of the method code but a direct branch to the code (see figure 6). This eliminates one instruction from the dispatcher code and saves space in the XMAS; however from a memory access point of view there is no difference (instead of the `ld []` in the XMAS the `br` in the VMT requires an additional memory access for fetching the branch target).

The 4-byte version of the `br` instruction can perform relative branches between -268435456 and +268435452, which effectively limits the LVM to a 256 MB contiguous address space. Since the LVM is mainly targeted for small embedded devices, this does not pose a real limitation. Another side effect of the unusual VMT usage is that the VMT of a class can no longer be created by copying the VMT of its superclass and changing and adding some slots. Since the `br` instructions are *relative* branches, the VMT contents also need to be relocated.

There is one important difference between XMAS blocks and the classic JVM method block: for one and the same method implementation multiple XMAS blocks might exist, because an inherited method implementation is referred to as a member of different classes (see example in section 5.3: method `B/one()V` can be invoked as `B/one()V` but it can also be invoked as `A/one()V` when an object of class `B` is manipulated through a reference of type `A`).

All information that is not stored in the XMAS block is, in fact, appended to the method code. A "magic" number marks the end of the actual code and the beginning of the method's stack frame descriptor, exception table, line number table and other attributes.

Similar to the JTOC data structure of Jalapeño [A+00], the LVM has one hashtable that contains pointers to the class structures of all loaded classes, using the fully-qualified class names as keys.

Each of the class structures has another hashtable containing XMAS pointers for all methods of that class. The name-and-type pairs [LY99, §2.10.2, 4.4.2] of the methods serve as keys for those subtables.

The hashtables are only accessed when classes are loaded or methods are resolved, for example, to change a particular XMAS from unresolved to devirtualized state. For regular method invocations of any type the hashtables do not need to be accessed.

5.5 XMAS for Fields and Static, Special or Interface Invocations

Not only `virtual` method invocations are translated into a call to an XMAS, but also `special`, `static` and `interface` invocations. Even the access to static and nonstatic fields is handled through an XMAS.

Static and special invocations (JVM instructions `invokestatic` and `invokespecial`) use the same XMAS layout as virtual invocations for the "unresolved" and "devirtualized" stages. The difference is that they remain in that state and never get revirtualized. Also, they use different entry points into the method body (figure 6), so that they are not affected by revirtualization that happens through binary rewriting (see section 5.6).

The `invokespecial` opcode is used for calling constructors, private methods and methods of the direct superclass [LY99, §6].

When referring to `private` or `super` methods, or constructors of the superclass or the class itself, `invokespecial` does not use an XMAS. As the superclass of a class always has to be loaded and resolved first (see 4.2), those invocations can be translated into direct calls because their target address is already known at that time. Only invocations of constructors whose classes have not been loaded yet are indirected through an XMAS.

Interface method invocations (`invokeinterface`) use the same format as all other invocation types in their unresolved state. When they get resolved, the XMAS contains code that branches to a special interface method dispatcher. This dispatcher looks at the actual type of the object on which the method is being invoked. The dispatcher then uses a lookup

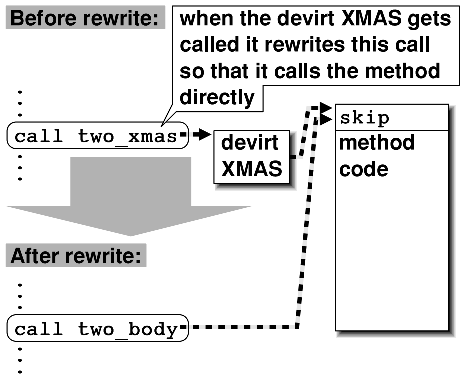

Figure 7: Turning an indirect call (through the XMAS) into a direct call with binary rewriting.

table to retrieve the interface method table (IMT) for that particular object type. After that, the IMT is used similarly to a VMT.

The JVM instructions `getfield` / `putfield` and `getstatic` / `putstatic` are also translated with the help of an XMAS. In the unresolved state the XMAS forces loading and resolution of the affected class. After resolution, the XMAS leaves the absolute address of the field on top of the operand stack. Read access is simply done with an `ld []` instruction, write access with `st []` and `pop`.

5.6 Binary Rewriting

Binary rewriting is a form of self-modifying code and is known as an optimization technique for a long time. Even so, it is no longer used very often. The reason for this is that most modern microprocessor architectures have large instruction caches, which are very problematic in conjunction with binary rewriting. For example, if self-modifying code is used on a StrongARM SA-1110 processor, a cache synchronization is required after the modification took place. The penalty for such a cache synchronization operation can be in the order of 10000 instruction cycles.

Being a very cheap and simple microprocessor, the IGNITE's only instruction cache is the 4-byte "cache" of its instruction pre-fetch. There is no other instruction cache and therefore there is no synchronization problem when binary rewriting is used.

One of the ideas behind the design of the LVM was that, if a Java method can theoretically be invoked with one single direct branch, the LVM should be able to perform the method invocation with a single branch.
In other words, if the detour through the XMAS turns out to be unnecessary the LVM should use a direct branch.

The XMAS for devirtualized methods calls a routine named `makeshortcut`. This routine transfers execution to the address that is stored in the second word of the XMAS.[‡)] Before actually jumping there, it modifies the original `call` instruction in the calling method, so that it now calls the revirtualized ("revirt") entry of the target method directly. Figure 7 visualizes this process.

When the runtime system detects that a devirtualized method must be revirtualized, two things will happen: first, the XMAS of that method will be changed into the "revirtualized" form and, second, the first word of the method code (which originally just contained a `skip` instruction) gets rewritten with a call to a revirtualizer routine ("revirt code"). Any call sites that enter through that first entry point will also get rewritten to the indirect form which branches to the XMAS instead of directly going to the method body. Figure 8 visualizes these steps.

5.7 Lazy Argument Passing

The differences between the argument passing mechanisms of the JVM and the IGNITE make it necessary to move method arguments from the IGNITE operand stack to the IGNITE local variable stack.

However, it can be observed that a lot of Java methods start with the instruction `aload_0` or a sequence like `aload_0`, `iload_1` (or even `aload_0`, `iload_1`, `iload_2`).
Especially, the `aload_0` at the beginning of a method is very common, since this instruction is the bytecode equivalent of a "`this`" reference.

When a method begins with an instruction sequence that fits into this scheme, it effectively just moves data back to where it was before. Thus, during argument passing, the arguments do not need to be moved to the local register stack in the first place and the instructions at the beginning of the method can be optimized away.

One important prerequisite for this optimization is,

[‡)] because the routine is called with `call` instead of `br`, the address of the subsequent word is stored in `r0` as actual return address; `makeshortcut` removes that address from the stack afterwards

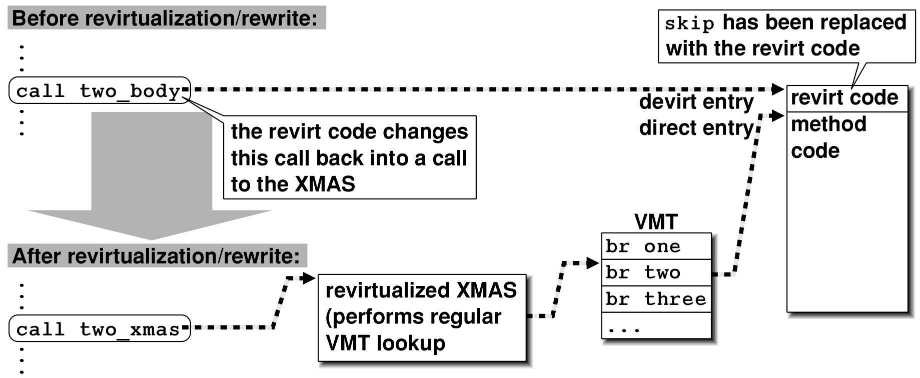

Figure 8: Revirtualization process for methods that were using a "shortcut" created with binary rewriting

Vars optimizable:	0	1	2	3
static				
PJava 3.0.2 classes	66.74	24.43	6.45	2.38
KVM Ref. Impl.	69.03	24.73	5.41	0.83
Kaffe `klasses.jar`	58.58	30.20	9.04	2.19
SPECjvm98	71.54	23.07	3.74	1.66
dynamic				
SPEC 200check	60.81	38.53	0.53	0.14
SPEC 201compress	60.16	39.64	0.15	0.05
SPEC 202jess	61.39	37.89	0.58	0.15
SPEC 227mtrt	31.09	65.82	0.74	2.36

Table 7: Optimizable Methods (static and dynamic)

that the local variables which are optimized are not used elsewhere in the method. So, in summary, the criterium is:

- The method starts with *x*`load_0` [, *x*`load_1` [, *x*`load_2`]] (in that order)
- There is no other read access of the local variable(s) 0 [, 1 [, 2]] except at the beginning of the method
- Instruction 0 [, 1 [, 2]] is not the branch target of a `goto`, `if`*cond* etc.

Already in the static analysis of the classes.zip file for PJava and the KVM reference implementation (table 7, line 1 and 2), it becomes obvious that optimizations of three variables can only be done in very rare cases. However, for about 9% of the methods found in the klasses.jar of the KaffeVM an optimization of 2 variables is possible.
Unfortunately, the dynamic view (table 7, lower part) looks different. The dynamic results for various SPEC jvm98 benchmarks were weighted with the number of invocations that actually happened at runtime. The tests yielded that only optimizations of one variable will make a significant difference. However, depending on the benchmark, between 38 and 65 (!) % of the method invocations benefit from an optimization. However, the actual degree of optimization varies greatly between different types of applications.

The LVM uses a special argument passing scheme (called "lazy argument passing") to apply the above optimizations. For methods that have 3 or less arguments (`this` counts as an argument, too) all arguments are left on the IGNITE operand stack. It is the responsibility of the target method to move the arguments to the local registers – or leave some of them on the operand stack, if the optimization can safely be applied. The LVM AOT compiler can statically decide how many variables can be optimized, if any at all. If possible, it will produce code that applies the optimizations.
If a method has more than 3 arguments, the calling method moves arguments 4 to n to the local register stack. The first 3 arguments are left on the operand stack.
The above scheme has another inherent advantage: The IGNITE microprocessor can only access the top three operand stack registers (`s0`, `s1`, `s2`). In the original JVM method invocation scheme, the object reference, which determines which method is the correct receiver of an `invokevirtual`, is at the bottom of the operand stack – buried deep below all the additional arguments. With the LVM lazy argument passing scheme, the object reference can easily be copied to the top of stack by issueing a `push s`p (where p is 0, 1, 2; see also table 6).

5.8 Translation Example

A simple example can illustrate how the AOT compiler of the LVM translates method invocations.
The example shows the translation of the `write([CII)V` method in class `java/io/BufferedWriter`. The method has 4 arguments (the implicit `this` argument and 3 additional arguments). An `invokevirtual` of this method is translated as follows:

```
pop lstack     ; move 4th argument to LR stack
push s2        ; get object reference
skipnz         ; check object against null and
call /throw_s2 ; trigger exception if necessary
call xmas...   ; execute XMAS of method
```

It is the responsibility of the called method to establish the stack link, move the first three arguments to the local register stack (if necessary) and allocate space for additional local variables (if necessary).

6 Conclusion

Currently, the LVM is still lacking most of the standard CLDC class libraries, which is the reason why only a limited number of benchmark results for the outdated Embedded CaffeineMark suite is available. In comparison to the earlier PJAE port, the score in the ECM3 "Method" test was increased from 50 to 138. The "Logic" score was 187 instead of 96, and the "Sieve" score went up from 128 to 205 (all results were measured on a 100 MHz IGNITE NC1 reference board). Even though the LVM is still far away from a complete JVM/KVM product, it proved that especially the optimized method invocation scheme is advantageous to the IGNITE architecture.

The IGNITE I processor is a "softcore" described in VHDL, which means that new instructions can be added and tested easily. A netlist for Xilinx Virtex FPGAs and other common FPGA devices can be freely downloaded from the PTSC website (`http://www.ptsc.com`) for evaluation.
Possible near future developments of the processor are additional instructions for null pointer testing, stack frame linking and additional loop instructions (for `ifle`, `ifgt` etc.).

References

[A+00] B. ALPERN / OTHERS. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211 – 238, 2000.

[DGC95] J. DEAN / D. GROVE / C. CHAMBERS. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the ECOOP'95 Conference*, 1995.

[Dic01] D. DICE. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 79 – 90, Monterey, CA, 2001.

[GJSB00] J. GOSLING / B. JOY / G. STEELE / G. BRACHA. *The Java Language Specification.* Addison-Wesley, Reading (MA), 2nd edition, 2000.

[LY97] T. LINDHOLM / F. YELLIN. Java Runtime Internals. Presented at the JavaOne'97 Conference, Track 1, Session 27. San Francisco (CA), 1997.

[LY99] T. LINDHOLM / F. YELLIN. *The Java Virtual Machine Specification.* Addison-Wesley, Reading (MA), 2nd edition, 1999.

[MMBC97] G. MULLER / B. MOURA / F. BELLARD / C. CONSEL. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, Portland (OR), 1997.

[PTS00] PTSC, San Diego, CA. *PSC1000A Microprocessor Reference Manual*, August 2000.

[Ran99] M. RANER. Implementation der Java Virtual Machine (*in German only*). *Java Magazin (Germany)*, pages 34 – 40, issue #6.99, 1999.

[Ran00] M. RANER. Teaching Object Orientation with the Object Visualization and Annotation Language (OVAL). In *Proceedings of the ACM ITiCSE 2000 Conference*, pages 45 – 48, Helsinki, Finland, 2000.

[TMH99] J. TRYGGVESSON / T. MATTSSON / H. HEEB. Jbed: Java for Real-Time Systems. *Dr. Dobbs Journal*, 24(305):78 – 86, November 1999.

[WG98] D. WHITE / A. GARTHWAITE. The GC Interface in the EVM. Technical Report SML TR-68-67, Sun Microsystems Laboratories, 1998.

[Win99] WIND RIVER SYSTEMS, Alameda, CA. *Personal JWorks Programmer's Guide 3.0*, 1st edition, May 1999.

[Yel96] F. YELLIN. The JIT Compiler API. URL: `http://java.sun.com/docs/jit_interface.html`, 1996.