ESPRIT LTR 21917 (Pegasus II)

Deliverable 4.1.2

Java Support Report

Stephen Early
University of Cambridge

September 1997

1 Introduction

This document describes initial work in supporting a Java Virtual Machine (JVM) on Nemesis. A general introduction to the internals of the JVM will be given, followed by a discussion of the various issues that arise during a port.

Initial work on Java under Nemesis was performed using Kaffe [4], a clean-room reimplementation of the Java Virtual Machine specification available under the GNU General Public License. Over the course of the investigation, two ports of Kaffe to Nemesis have been performed.

Recently Cambridge University Computer Laboratory has obtained a license from Sun to use their Java Development Kit source code. There are distribution restrictions on JDK ports[1], but binaries can be made generally available. An initial look at the JDK indicates that most of the issues described will be the same, but there may be differences and the internal interfaces[2] of the Sun code may constrain the way that the port is done.

2 The Java language

Java is a general-purpose, concurrent, class-based, object-oriented language. It is relatively high-level, including garbage collected storage management, and excluding unsafe constructs such as array accesses without index checking. The Java language is specified in [1].

Although it is possible to compile Java to a native machine code, it is usually compiled into 'Java Bytecode'. This is then interpreted at runtime by the JVM. As an optimisation, the JVM can compile the bytecode into native machine code at runtime (a process known as 'just in time compilation').

Java bytecodes are stored in 'class' files, which have a strictly defined format. The JVM does not deal with the Java language at all; it only works in terms of class files and Java bytecodes.

Classes are grouped into 'packages', which are named hierarchically. For example, a package written at the computer laboratory in Cambridge may be named `UK.ac.cam.cl.foo`. Classes within the same package may have privileged access to each other's methods and fields.

A standard set of APIs have been defined, which consist of packages whose names begin with `java`. The core API, which all useful Java programs have to use, consists of the packages `java.lang`, `java.io` and `java.util`. Other packages define higher level parts of the API, for instance `java.awt` covers graphics and windowing.

The combination of the JVM and the Java API is referred to as the 'Java Platform'. The goal of Java is for a program compiled for one implementation of the Java Platform to be runnable on any other implementation as well.

3 Java Virtual Machine

[1] Only diffs against JDK source code may be published, not the whole source code.

[2] Sun expect that people porting their JVM do so by implementing a set of functions specified in a header file. They expect people *not* to have to change any of the architecture-independent code, and strongly discourage such changes.

The Java Virtual Machine is an abstract computing machine that reads files in the 'class' file format. These files contain Java bytecodes, a symbol table, and other useful material like debugging information. The JVM is specified in [3].

When the JVM is started it performs internal initialisation (which may involve allocating memory for a heap, creating 'daemon' threads to deal with garbage collection and finalisation, etc.) and finally creates a single thread in which it runs the `main()` method of a specified class. The virtual machine continues running until there are no non-daemon threads left, or until the `java.lang.Runtime.exit()` or `java.lang.System.exit()` method is called.

4 Memory management

The JVM requires memory for two purposes: storage of objects, and stacks for threads. In most JVM implementations, a heap is used for both objects and thread stacks. This has the disadvantage that stack overflow checking must be performed explicitly by the code responsible for setting up a stack frame; if a separate stretch[3] were used for a stack then overflow checking could be done by memory management hardware. The advantage of performing this check is that another segment of stack can be allocated if required.

The JVM heap could be implemented on Nemesis in several different ways. The options are:

1. Use the existing Nemesis `Heap$Malloc()` and `Heap$Free()` calls to obtain memory in a heap that is shared with other Nemesis code.

2. Use `Heap$Malloc()` and `Heap$Free()` as before, but use a heap that is dedicated to the JVM.

3. Implement a Java-specific heap manager inside an area of memory obtained through `Heap$Malloc()`. If the JVM heap turns out to be too small, more memory can be obtained through another call to `Heap$Malloc()`.

4. Implement a Java-specific heap manager in a stretch of memory obtained specifically for the purpose. Currently the size of the stretch would correspond to the amount of memory allocated for the heap. Eventually, once the Nemesis memory management system has been developed further, a very large stretch of virtual address space can be allocated and the heap manager can also manage the swapping of pages into and out of the stretch.

The first port of Kaffe used the first option, mostly because Kaffe at that time didn't have its own heap manager. This worked, but was obviously not ideal. The second port of Kaffe used the fourth option. This was made easier because by the time of the second port Kaffe had its own heap manager, designed to work using blocks of memory obtained using `mmap()` under Unix.

The Sun JVM implementation has its own heap manager. In fact, there are several variations which cover most of the memory allocation scenarios that will be found on all platforms, from using `malloc()` and `free()` to using raw physical memory. The one that looks the most promising for use on Nemesis is usually implemented over `mmap()`, and understands

[3] Area of virtual memory.

paging. We can implement this over a raw stretch on Nemesis, and it should work quite efficiently once we are are able to do our own paging.

5 Built-in classloader

Every class that is loaded into the JVM is loaded by a *classloader*. Classes are identified uniquely by the (classloader, classname) pair. It is possible for two classes of the same name to exist in the JVM at the same time as long as they are loaded by different classloaders; this mechanism helps separate programs in the JVM. For example, it is possible to implement different security policies for classes that are loaded by different classloaders.

From the point of view of a Java program, a classloader is a subclass of a standard system class, `java.lang.ClassLoader`. However, during JVM startup no classloader classes are available; they must also be loaded. To enable the JVM to start, and thereafter to load 'system' classes and standard libraries, the JVM includes a classloader. This fetches class definitions from some system-defined place. In Unix-based implementations of the JVM this place is usually defined in terms of a *classpath*, a colon-separated list of ZIP files[4] and directories to search for classes.

This approach to specifying the standard place to look for classes will not work under Nemesis. There is no single filesystem; any number of filesystems may be mounted, and different domains may have different sets of filesystems available under different names. There is currently no way of specifying a filename on a particular filesystem as a string. It is possible to specify a file or directory as a (filesystem,filename); this will yield a File or Directory closure.

The approach taken in the first two ports of Kaffe was to pass in a Context, the entries in which were File or Directory closures. When looking for system classes, the built-in classloader would examine each of the entries in the Context in turn. If a File was found it would be assumed to be a zipfile, and the class would be searched for in the zipfile. If a Directory was found then a pathname relative to that directory (for example `java/lang/Thread.class`) would be constructed and if present the class would be loaded.

The code that is run to load classes once they are found had to be re-written to use Nemesis 'Rd' and 'Wr' interfaces rather than the Posix-style 'read' and 'write' calls. This change was fairly trivial.

6 Threads

Most Java programs start multiple threads, so an efficient thread system is vital to the good performance of the JVM. JVMs have been implemented on a wide variety of platforms, many of which do not provide a threads package. A lot of implementations of the JVM, therefore, include their own threads packages.

Different approaches to implementing Java threads were taken in the two ports that have been done to date. They are described in the sections below.

[4] A 'zip' file is a type of archive commonly used on DOS and Windows machines. It is used to store classes for the JVM for two reasons: it permits long filenames, and it reduces the inefficiency involved in storing many small files. Zip files can be manipulated using commonly available tools, both commercial and free. Although a common use of Zip files is compression, the JVM does not support the use of compressed Zip files.

6.1 Initial port

The aim in the initial port was to find out whether supporting a JVM was feasible, and to complete the proof-of-concept in the simplest way possible. I decided to ignore the thread scheduler code already present in Kaffe, which relied heavily on Posix-like signals and alarm timers. Instead I built the JVM on top of a standard Nemesis thread scheduler. This raised some interesting issues.

At the time the first port was done, we did not know how to rewrite code automatically so that it was sharable in a single address space. For Kaffe, I did this by hand by defining a state record and using macros to convert references to global state into references to the state record. The macros expected the state record to be accessible pervasively, so I extended the Pervasives[5] record to include per-thread and global Java state pointers.

The Pervasives of each thread that was to be able to run JVM code needed to be extended. At the time of the initial port I was uncertain how the JVM would fit in with the rest of Nemesis; would it be desirable for arbitrary C code to call arbitrary Java code, for example? To allow for this I needed to extend the pervasives of every thread in the domain, whether it was started within the JVM or not. This required the registration of some ThreadHooks—callback functions to report on the creation and deletion of threads.

The standard Nemesis threads package was missing some of the functionality necessary for the JVM. In particular, it did not have functions to suspend and resume threads, and it did not have any call to return the stack details of a thread. The latter is necessary so that the garbage collector can sweep thread stacks for references to objects; stacks are 'root' objects for garbage collection. I added a method to the `Thread.if` interface called `GetStackInfo()` which returned the address of the top of the thread's stack and the current value of its stack pointer.

The first port of Kaffe was unsatisfactory. On further thought I have decided that we do *not* need arbitary Nemesis code to be able to call Java methods directly. Instead we should be able to set up IDC servers in Java. (See section 11 for more information.) This simplifies the interface to the thread scheduler considerably; all that is needed is for the 'start of thread' function to modify its Pervasives appropriately, and undo the modification before returning. There is no need for any ThreadHooks.

For the initial port I did not modify the standard thread scheduler to include suspend and resume functions. I expect that this modification will eventually be made, because the functionality is useful even outside the context of a JVM port.

6.2 Second port

The second port of Kaffe took a completely different direction from the first. Instead of trying to integrate a JVM tightly with a standard Nemesis domain, the JVM was made into a completely different type of domain. This change was made out of a desire to use the thread scheduler from Kaffe as the thread scheduler for the domain; every thread that exists in the domain is a Java thread, and is started by a call to `java.lang.Thread.start()` in the usual way.

The implementation of this new type of domain was reasonably simple. In the time between the first and second ports of Kaffe the standard Nemesis thread scheduler was simplified; the code used to deal with activations and events was moved into separate modules

[5] For the importance of the pervasives on Nemesis see [2].

and given interfaces, making it easier to implement new thread schedulers without having to worry about the implementation of code to handle events on channels into the domain.

The worst problem with the implementation of this port was a circular dependency. In general it is necessary to have the thread system working in order to be able to access filesystems[6]. However, for this domain it is necessary to have loaded several classes (`java.lang.Object`, `java.lang.ClassNotFoundException`, `java.lang.ThreadGroup`, `java.lang.Thread`, etc.) before threads can be started.

The solution adopted was to create the JVM heap in the domain that is starting the JVM, and preload it with all of the classes required to start the thread scheduler. The JVM domain would then start up, initialise the thread scheduler and connect to the appropriate filesystems for loading further classes.

It is necessary to support the usual Nemesis native code interface to threads for some native code to be able to work; a call to `Threads$Fork()` should create a new thread running native code. A new class called `nemesis.Thread` was created to support this. A call to `Threads$Fork()` constructs an instance of this class, storing the address of the native code to call in it. It then calls the `start()` method of the object, which results in the creation of a new thread. The new thread calls the `run()` method of the `nemesis.Thread` object; this is a native method which jumps to the code previously specified.

6.3 Future ports

The Sun JVM has an internal interface for threads which has operations like `create()`, `init()`, `free()`, `exit()`, `yield()`, etc. This leaves the question of implementation open; we can implement our own internal thread scheduler, or we can work on top of an existing thread scheduler.

For the first attempt at porting the Sun JVM we intend to create a Nemesis thread scheduler that has semantics appropriate for a JVM; it should support all of the suspend/resume calls, as well as providing hooks for more complex things like enumerateStackFrames(), etc. It is necessary to be able to access the stacks and stored registers of threads in order to avoid garbage collecting objects that may only be referred to on a thread stack or in a thread's registers.

It might be appropriate to extend the standard Threads interface in this way and insist that all thread schedulers on Nemesis implement sufficient features to support a JVM. The code involved in doing this should be quite simple. The JVM uses priority as a policy for scheduling threads; however, it does not assume that the thread scheduler will take any notice of thread priorities. This is fortunate; priority is not always an appropriate model, and it would be bad for all thread schedulers to have to implement it.

6.4 Stack allocation

The two current ports of Kaffe simply allocate memory for thread stacks in a stretch or on the heap. No stack bounds checking is performed, and stacks cannot be extended. This has proved sufficient for all Java programs run so far.

The Sun JVM supports extensible stacks; stack segments are allocated on the heap, and the stack is checked for overflow at the start of every method invocation. This should enable

[6] For example, a filesystem implemented over TCP/IP will require a retransmission thread to be running.

less space to be allocated for stack on average, because the initial stack allocation for a thread can be small.

6.5 Contexts

The standard Nemesis thread scheduler allocates one 'context slot' per thread. The information stored in this slot is *either* a full context saved by the NTSC *or* a `jmp_buf`, these options corresponding to the case where a thread has been pre-empted and where a thread has blocked, respectively. This implementation is simple, but gives a fairly low limit to the number of threads that may exist. Since Java programs tend to create a lot of threads, a better implementation must be used.

It is possible to write a thread scheduler that will support an arbitrary number of threads while using only two VP-level[7] context slots; one for the NTSC to use as a 'resume' slot for when it has to preempt the domain with activations disabled, and one as a 'save' slot for preempting the domain when activations are enabled. The thread scheduler can copy the context out of the 'save' slot as soon as it is activated. This is obviously not a good solution.

A better solution is to use as many context slots as are provided as a cache of active thread contexts. Threads that are blocked do not need to have a VP-level context slot. Threads can be allocated a context slot when they are run, and can have it taken away again as soon as they block. It should only be necessary to take a context slot away from a running thread if there are more runnable threads than context slots.

7 Synchronisation

Every object has associated with it a lock and a wait set. Threads may compete to acquire the lock. Unfortunately the combination of lock and wait set do not have the same semantics as the SRC mutexes and condition variables already available in Nemesis.

According to the JVM specification, only one thread at a time is permitted to lay claim to a lock. A thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed.

It can be inefficient to create a system-level lock for each object, especially since most of the locks will be unused most of the time. Kaffe keeps a pool of mutexes and condition variables, which it allocates to objects and frees as necessary. A lightweight locking mechanism is used while doing this, which on Nemesis translates to the disabling of activations.

The Sun JVM has an internal interface to 'Monitors', with calls like Enter(), Exit(), Wait(), Notify(), etc.; this enables the port to be done without worrying too much about the exact mapping between objects and monitors. An implementation of 'Monitors' over event counts and sequencers should be straightforward.

8 Garbage collection and Finalisation

The two ports of Kaffe used its standard incremental garbage collector. This runs as a separate thread, and collects either when explicitly requested by a call to `java.lang.Runtime.gc()` or implicitly when the amount of free space in the heap becomes small.

[7] Virtual Processor.

The Finaliser also runs as a separate thread, which is kicked after each run of the garbage collector. It calls the `finalize()` method of objects about to be destroyed.

This scheme should work in most ports of the JVM. Synchronisation between the GC thread, the Finaliser thread and the rest of the JVM can be done using the standard Java synchronisation primitives. The only problem arises in the need to find *all* object references during garbage collection; some objects may only be referred to by variables held on thread stacks, or even in the registers of a blocked thread. Allowing the garbage collector to access these references requires support in the thread scheduler.

9 Native methods

Methods marked as 'native' in Java class files do not have a bytecode implementation. When they are called, a native routine is called instead. This usually done for one of two reasons:

- Performance; some routines (for example compression and decompression) work much faster when implemented in native code
- Performing special operations; methods that need to call the operating system, access memory directly, or call routines in the core of the virtual machine (defining a new class explicitly is one example) can only be implemented in native code.

Native methods are usually only used in classes that are part of system libraries. They are not generally used in applications, and are definitely not used in applets; if they were, it would not be possible to run the application on JVMs which did not implement the appropriate native method[8].

Native code is added to the JVM by a call to `java.lang.Runtime.loadLibrary()`. This call can only be made if the current security manager permits it, so arbitrary application and applet code cannot add its own native code to the JVM. The exact effect of this call is defined by the implementor of the JVM; it may make a JVM on Unix call the dynamic link library routine `dlopen()`, for example. This call is a no-op under Nemesis.

When the JVM encounters a native method it looks up the address of the appropriate native code and calls it. In implementations of the JVM on Unix this is often done using a library call like `dlsym()`. Nemesis does not currently have support for this kind of operation. The solution adopted in the two initial ports was to implement the `Context` interface such that looking up a method name in the context returns the address of the native routine for that method[9]. A `MergedContext` can be used to merge together a number of separately compiled sets of native methods.

Many of the issues which affect writing code for native methods on Unix (for example the fact that the code must fit in with whatever threading package is being used) are not a problem on Nemesis. Native method code can be written in exactly the same way as any other native Nemesis code.

Some native methods must be implemented in the same symbol space as the core of the JVM, because they are the interface between Java code and the JVM. Examples include

[8]Javasoft have published a specification called *JNI*, the Java Native Interface, which should allow native method code to work with different JVMs on the same platform and architecture. There is not currently a choice on Nemesis.

[9]Example: looking up `java_lang_Thread_start` returns the address of the implementation of `java.lang.Thread.start()`, a routine used to start a new thread.

the implementations of methods in `java.lang.Thread` (for starting and stopping threads), `java.lang.Runtime` (miscellaneous operations like requesting garbage collection, switching tracing on/off), and `java.lang.ClassLoader` (turning arrays of bytes into linked objects). These are compiled into the same block of code as the core of the JVM, and their Context is added to the native methods MergedContext by default.

10 Exceptions

Porting the code that deals with exceptions in Java bytecode does not normally require much attention. However, once the possibility of catching and raising exceptions in native code is introduced some more issues arise: how do we convert Nemesis exceptions to Java exceptions? How do we convert Java exceptions to Nemesis exceptions?

Catching exceptions is relatively simple. The real problem is in understanding them; Java exceptions are arbitrary Java objects[10] and as such have methods, public and private fields, and most importantly a type. Interpreting this in native code is not trivial.

A simple solution is for Java exceptions that reach a native part of the stack to be reraised as a `Java.Exception[object:ADDRESS]` in Nemesis. Native code that cares what the exception is can invoke a private Java method to interpret it. Nemesis exceptions that reach a Java part of the stack can be reraised as a `nemesis.Exception()`. Java code that expects this can call a native method to interpret it.

The final solution is probably going to be to do nothing except define that native methods will not raise Nemesis exceptions; native methods can be considered to be a part of the VM, and can be expected to behave properly and not allow Nemesis exceptions to traverse Java sections of stack. The implementation of the simple solution described above is up to the writers of native methods, and does not require extra support in the JVM. Native method authors who do not call Java methods from native methods will have no problem anyway.

11 Future work

The next step in supporting Java on Nemesis is to port the Sun JDK. This will provide a solid base for further work, and should enable Java on Nemesis to keep up with the large number of new Java APIs currently in preparation.

Porting the Sun JDK involves implementing a Nemesis-specific backend for the Abstract Windowing Toolkit (AWT). This task is becoming simpler; the design of the AWT is moving towards user interface components being rendered by Java code, rather than relying on a native widget set. The use of subwindows as a means of stacking control and event demultiplexing is also being reduced; the new 'lightweight components' do not require window-system level subwindows.

The Java Foundation Classes, some of which are available now and some of which will be released in version 1.2 of the JDK, should provide a rich toolkit for building user interfaces on Nemesis. It should be possible to implement the Java Media API extremely well on Nemesis, allowing multiple streams of continuous media to be managed by portable Java programs. This portability will allow a side-by-side demonstration of multimedia on Nemesis and other operating systems.

[10] Provided that they are subclasses of `java.lang.Throwable`.

`middlc`, the program used to interpret MIDDL interface files, could be extended to produce Java language stubs. This should enable Java programs to call arbitary Nemesis services. Alternatively, an ORB[11] could be implemented on top of the JVM that is able to invoke operations on Nemesis-hosted servers, and which can host servers that may be invoked transparently from native Nemesis code.

12 Conclusion

It is possible to host a Java Virtual Machine on Nemesis. There are many possible approaches to the implementation, two of which have been tried. Work to date has been based on Kaffe, a clean-room implementation of the JVM specification that is available under the GNU General Public License.

The standard Nemesis thread scheduler cannot support the JVM directly. A JVM implementation must provide a thread scheduler that has appropriate semantics. It may be desirable to extend the standard Nemesis thread scheduler so that it is appropriate.

Future work on Java under Nemesis will be based on a port of the Sun JDK to Nemesis. Work on this port will begin shortly.

13 Acknowledgements

Some of the work on the second port of Kaffe was done at the 'Nemesis Festival' in Cambridge in July 1997 by Robin Fairbairns and Rolf Neugebauer.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN 0 201 63451 1.

[2] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. ISBN 0 201 63452 X.

[4] T.J. Wilkinson and Associates. `http://www.kaffe.org/`.

[11] Object Request Broker