CSRP 499
Implementing a Java Virtual Machine for Network Simulation

Rory Graves and Ian Wakeman *†

October 1998

Abstract

In this paper we discuss the development of a controllable virtual machine for use within network simulations. We describe general problems in integrating virtual machines with simulation environments, illustrated through our experiences in building a Java Virtual Machine.

1 Introduction

A burgeoning area of network research is in Active Networks [11]. In active networks, packets contain both date and *code*, which can be executed on intermediate switches. In order to evaluate the effectiveness of this decision, simulations of network algorithms and protocols need to be able to measure the effect of processing load on switches as well as on the more traditional resources of buffers and bandwidth. We have thus designed a simulation environment building upon the *ns* network simulator [8], in which we integrate virtual machines on which to run the packetised code.

Most Active Network research uses the Java Virtual Machine as the computational substrate. We therefore decided upon a Java Virtual Machine (JVM) as our initial target machine.

Many JVMs already exist (see section 2.1) but we have found none that are suitable for simulation work. This is because JVMs are written for speed, not clarity or controllability. To use a JVM in simulation requires the ability to "step" the JVM and keep it synchronized with "simulation time", and to allow arbitrary instrumentation of code within the JVM. We also need to experiment with various choices within the design of the JVM, such as the scheduler, the class loader and the garbage collection system, requiring the JVM to be amenable to accepting new implementations of these services.

*This work is sponsored by EPSRC grant GR/L06072, BT Laboratories and Hewlett-Packard.

†School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, UK

In the rest of this paper, we first describe our requirements in detail, and describe how other implementations of JVMs are inadequate. We then outline the major problems in building our JVM, and how we have overcome them, concluding with a discussion of how the JVM is used within our simulation environment.

2 Virtual Machine Requirements

A machine that is steppable The VM should be controllable down to the level of individual instruction execution. We must define an indivisible unit of time (a step). Some instructions may take more than one step to execute (the more complex instructions). The machine should be controllable at the level of these steps.

Ability to assign arbitrary periods of simulation time to instructions You should be able to ask the VM to run for a given period of time (for discrete event simulation). The VM should then run until either the time period expires or the VM generates an external event that needs external scheduling or processing (such as sending a message).

Loosely coupled scheduling, garbage collection, memory management and class loading services, which are substitutable. The design of the VM should be very modular allowing different implementation of all of the major modules. This allows for experimentation with different algorithms which affect execution (such as garbage collection). Each module should have a well defined interface for interacting with other modules so they can be easily re-implemented and swapped.

Because we are not overly concerned with speed we concentrated on ease of programming and clarity. For this purpose Java was chosen as the implementation language. This also helps in several other ways. Floating point numbers often cause a problem because they must obey the IEEE 754 floating point standards. By using Java as the implementation language we already had IEEE 754 compliance. Making the JVM steppable caused more problems, this will be discussed in section 7. Testing and proving the JVM was more troublesome than at first thought; this is discuss in section 14.

2.1 What already exists

There are several good free implementions of Java. Sun provide the original at [9] and a cleanroom freeware implemention is available from KAFFE [7]. The source code for KAFFE is included and the source code for Sun's implementation can be downloaded from [10] after you sign Sun's usage policy document. The Sun source release is not much help as it is many lines of C and assembly presented *as is* without any supporting documentation.

There are also several commercial implementions available and the list is changing all of the time. None of the commercial implementations provided any

useful information so were ignored. The only exception to this was a vaguely interesting document from Ameran detailing their testing process and will be discuss in section 14.

Apart from these we found two implementations [2] and [12] which were implemented as MSc projects. Neither group had a complete implementation but provided lots of implementation notes within the project reports.

All of the implementations which we found were written in a mixture of C and assembly language for speed. None of them were controllable in the ways needed for simulation work but they did provide large amounts of useful information and a good idea of the project size.

The specification of the JVM has been made publically available by Sun in both book form [3] and on their web-site [4]. We are aware of two other books. The first [5] is basically an imitation version of the Sun book. These first two books seem to be light on "how to" information. The Sun specification regularly steps round sticky issues by citing them as "implementation specific" and ignoring them. The O'Reilly book has a small section of implementation notes. The other book is by Bill Verner [13] and contains a lot of useful information and examples.

3 Basic Architecture

The basic architecture of a JVM is well covered in [5] p51-62 and we will only cover the salient features briefly here. More detailed discussions are given where necessary in the relevant sections.

The heart of the JVM is the execution engine. This is responsible for actually executing the bytecode instructions. There is a large chunk of support code to deal with more complex operations such as method calls, exception handling etc. The execution engine interacts with a garbage collected heap and a class data area. The class area contains class and method information. This area may actually be part of the heap and may also be garbage collected. The class must interact with a verifier and sources of class information such as the network and the local file system. Another module is normally responsible for loading native method code from the local system.

4 Instruction Set

The basic instruction set is well defined in [3] and [5]. We make comments below on areas in which particular care must be taken.

4.1 Floating Point Numbers

All the floating point instructions in the JVM must comply with the IEEE 754 32 and 64 bit floating point standards. In most modern languages such as Java this is not a problem. When the standards are not supported by the implementation language care must be taken to ensure correct behaviour.

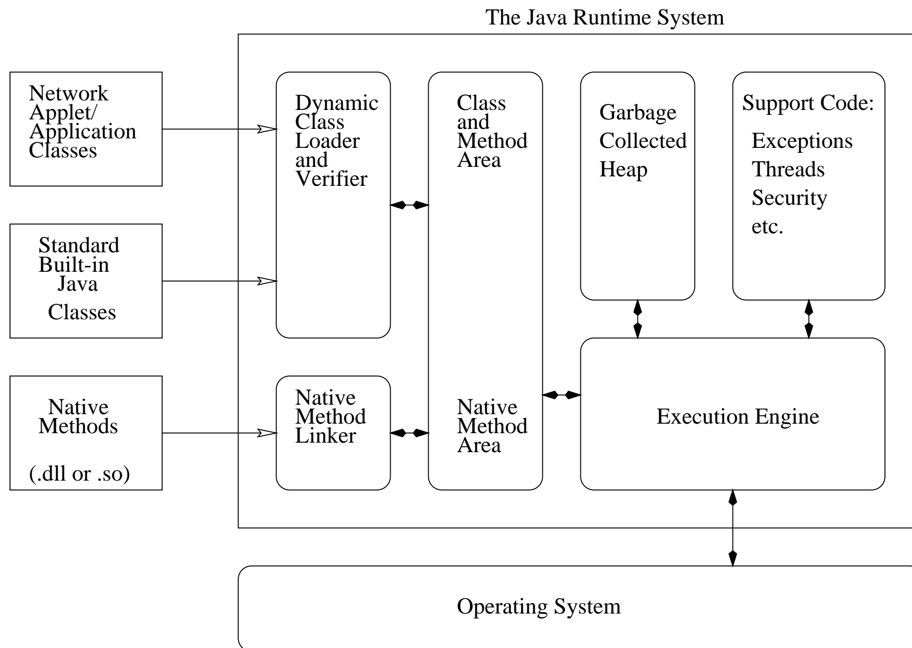

Figure 1: The Java Virtual Machine

4.2 Layout of class files

The class file layout is fairly straightforward, based on a table within table approach. However, a fair amount of work must be done by the VM for construction of virtual method tables etc. If the class file laid out this data in a better way class loading and setup would be made much easier. See the brief description of virtual methods in section 6 for a discussion of the missing information.

5 Classes and Objects

The loading of classes and their structure are well-defined in [3] and [5]. However, since the loading of classes is a complicated operation, there are a few unclear points specific to the JVM.

- The only feature that we found not clearly marked was the fact that long and double entries take up two entries within the table (the second entry being considered invalid).
- The system class has an undocumented feature. Instead of setting up the variables that it needs though a static initialiser, it contains a method called `initialiseSystemClass()` that must be called by the runtime environment after the class has been initialised otherwise the standard input and output streams are not configured.

6 Stacks and Locals

For faster execution the stack has been designed to always use 32 bit words. 64 bit quantities such as doubles and longs are handled as 2 32 bit quantities. The semantic rules state that a double word cannot be split. It is normally left to the verifier to check rather than the actual operation. This removes redundant runtime checks from the stack improving efficiency. Efficiency is discuss further in section 12.

The stack actually serves two purposes; as well as being the operand stack it also acts as the local variable area. When a method is called its arguments are pushed onto the stack. These arguments then form the beginning of the local variable area for the next stack frame. Long and double variables taking two 32bit variable slots as with the normal stack. The stack must be extended to ensure space for the whole of the locals area. (The size of the locals area is part of the method declaration in the class file).

Virtual method calls were slightly troublesome. The method arguments are placed on the stack with the object on which to call as the first argument (It becomes the first local variable as a 'this' reference). The obvious place to put the arguments stack size was within the method declaration. This is fine but causes a chicken and egg problem. Without knowing the actual method you don't know where the target object reference on the stack and without knowing the target reference you don't know which method is being called.

We solved this by adding extra information into the constant pool method reference. From the signature you can work out the number of stack words down the stack the new local area will begin. We broke it into three stages:

1. From the signature declare the new local variable area and make it the same size as the number of arguments.
2. Use the first local as a target object for a virtual method lookup.
3. From this information expand the local variable area to the needed size.

Stack operations are now pushed onto the stack above this area as normal. One problem causes here is on a failed method call the stack must be undone carefully so that no junk is left on the stack after a `NoSuchMethodException` or similar is thrown.

7 Threads and Scheduling

Multi-threading is the ability for a program to have several paths (or threads of execution) through a program executing in parallel. Threading has normally been seen as an extension to a language (such as the threads packages available in C). In Java multi-threading has been built into the language, with a set of classes and synchronisation constructs built in. This is great for the Java programmer but is a major headache for the JVM implementor. [6] is a good source of information about threads in Java.

Java provides a very simple mechanism for programmers to synchronize threads. Each object in Java has an associated lock. A thread may attempt to gain the lock on an object. If the lock is already held by another thread the thread is forced to wait. Synchronization can occur in three places. Firstly there are two JVM instruction `MONITOR_ENTER` and `MONITOR_EXIT` (generated by synchronized blocks). Secondly synchronized methods implicitly call these instructions when the method is entered and exited. Lastly synchronization also occurs with class loading. If a thread begins loading an initialising a class when another thread is already doing so it is forced to wait. The second thread is restarted and given the requested class when the first thread has finished initialising it.

Within this JVM there is no way for a separate clock to interrupt a thread running. Instead the JVM asks the scheduler to pass a certain amount of simulation time. This translates into the number of steps that can occur in a given period of simulation time (see 7.1). A thread will run for a given number of steps (a certain time period) just as a normal system. The scheduler is currently a fairly simple prioritised round-robin scheduler. Each thread is allowed to run for a defined number of steps before control is handed to another thread. This mimics reality fairly well.

The scheduler is also responsible for dealing with locks and synchronization issues. The lock manager and scheduler interact but are designed in a modular fashion. Each makes requests to the other as needed to add or remove threads from the running queues.

Little is specified about what the normal scheduling method is. It is declared "implementation dependent" and ignored. This gives the implementor freedom and flexibility so that an appropriate algorithm for their needs can be used. As stated the scheduler is currently a prioritised round robin scheduler but it is interchangeable with any other required scheduling module. There would be no problem changing the current implementation for a different scheduling algorithm.

7.1 Stepping

We required a large degree of flexibility in step control and granularity. We first considered using a finite state machine over the instructions but the ensuing complexity made this infeasible. The alternative design used interlocking threads. To run an instruction on the JVM the control thread wakes the VM thread and goes to sleep on a common lock. The VM thread executes a single step defined by us, wakes the control thread and then goes to sleep on the common lock. The control thread now continues execution possibly reawakening the VM thread if required.

Each simulated virtual instruction corresponds to a large number of real instructions inflicting a performance loss of several orders of magnitude. Whilst this does not impact process size or the actual simulation it does make the JVM simulation very processor intensive.

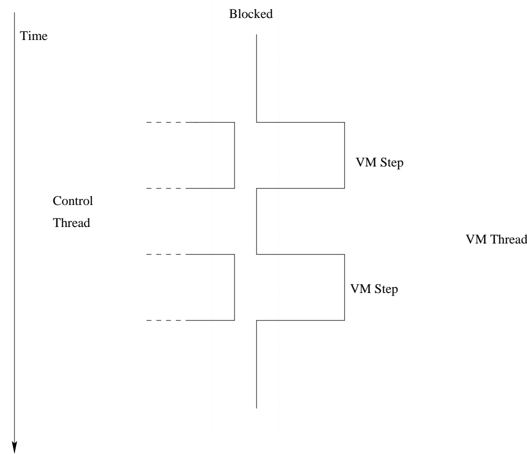

Figure 2: Thread interlocking for stepping

8 The Heap

True simulation of the heap has been ignored in this implementation for several reasons. Java is strongly typed which means that it is hard to use a block of memory to represent a flat memory area. The only obvious way visible to achieve this is to use a byte or int array as a heap and do data conversions to transfer to and from the represented types as needed. Although possible it wipes out the little efficiency that the implementation has with little benefit. It is beyond the scope of this project to explore much of these issues and so it was ignored.

To overcome the problems the heap is modelled through a lookup table. This creates a level of indirection. The table points to a storage object representing the object in the runtime environment. It is possible to add memory size checks to insure that a fixed size memory model can be achieved quite simply. The loss here is the simulation of fragmentation and other features that memory management is prone to. This would not pose a major problem with an indirection table as data can be moved without affecting the running program but still cannot be modelled within the JVM.

9 Garbage collection

GC is the process of finding and removing any object that can no longer be reached by the running program and then reclaiming the memory that was used for reuse later on. Below we discuss the basics of garbage collection and look at some of the problems. Both [14] and [1] as much fuller introduction this area.

Stop and collect garbage collection where the JVM is halted to do a garbage collection. This normally is only done when the JVM runs out of memory. This model is not suitable for the uses we plan to put the JVM to and so we implemented an incremental garbage collector.

Reference Counting For reference counting each object has a counter associated with it. This counter represents the number of different references to the object. Whenever a reference within the running program is modified the counters for the affected objects are modified. For example, if a reference is duplicated on the stack the reference counter would be incremented. When a reference is written over or destroyed the counter is decremented. When the counter reaches zero the object can no longer be referenced from the running program and can thus be garbage collected.

The only place that this algorithm falls over is when we have a cyclic data structure. If we create two objects A and B which refer to each other. When the running program can no longer reference each other the reference count is not zero. This is because A still holds a reference to B and vice-versa. Reference counting must be supplemented with another algorithm to guarantee that all the garbage is collected.

Mark and Sweep The concept behind mark and sweep garbage collection is simple. Take all of the reference on the stack and the local variable area. We will call these the root set. From each root object we take all of the references it contains and visit these recursively. As we travel through each object we mark it as 'visited'. Any object that is not marked as 'visited' when we finish is not reachable from the running program and can thus be garbage collected.

9.1 Problems

The two main versions of simple garbage collection detailed above have some problems. Reference counting, as described, cannot deal with circular references and must be supplemented with some other garbage collection method. Simple mark and sweep is fine as long as nothing changes during the collection. This means that the program must stop for garbage collection to take place. In many cases this is unacceptable.

9.2 Incremental Garbage collection

Incremental garbage collection allows collection during program execution. i.e rather than collecting when there is no more memory, collection can be done 'on the fly'. The most common technique for this is tri-colour marking as described below.

9.2.1 Tri-Colour Marking

Tri-colour marking works in a very similar way to a normal search. Every object on the heap has one of three colours:

White Objects that have not yet been visited.

Grey Objects that have been visited, but whose children have not all been visited.

Black Objects that have been visited along with all their children.

All objects start white, as they are visited they are changed from white to grey. When all of an objects have been visited they are coloured black. When there are no grey objects left all of the white objects left are unreachable and therefore may be garbage collected.

For this to work we must maintain two invariants:

1. No black object points to a white object
2. All grey objects are in a list of objects yet to be explored

The running program may still be creating and modifying objects. To keep the invariants valid we may recolor different objects so that the invariants hold. Different versions of the algorithm use different colouring mechanisms to do this.

9.3 Does no memory mean no memory?

When a garbage collecting memory management system reports that it is out of memory, it may not be accurate. If resource became free during the current pass the GC algorithm may not realise until the completion of its next GC cycle. This raises some interesting issues about when, and how much time should be spent garbage collecting. It may be better to allocate a percentage of the processor time to garbage collection rather than waiting to run out of memory first. The issues raised about when and how much garbage collection are a complete research issue in their own right and will not be explored here.

10 Native Methods

Native method handling is quite important in the implementation of a JVM. In the current implementation a set of native handler classes exist. Each one registers itself for a given class and supports all of the native methods for it. We have completed a partial implementation similar to the JNI (but a bit more object-oriented) which supports all of our current requirements. Currently all of the classes are loaded by the JVM at startup. This has an implication for user native methods as detailed below.

There are quite a few native methods in the API. Anything that can be seen as "implementation specific" is normally a native call to let the JVM handle it. One of the biggest problems with this is the lack of documentations for these methods which are normally internal to the class. Some of these are obvious from their name such as `yield0()` which is the internal method for the Thread.yield() system call. Others are marked as "helper functions" with no supporting documentation. We have only implemented the sub-set of methods needed to provide the facilities that for use in simulation. For example, most of the file handling and AWT system calls have been ignored.

We only partially support the calling of native methods by user programs.. The user native handling code must support the same interfaces as the internal

native handlers. Dynamic loading of these classes is not supported. This means that the JVM must be recompiled to include the handler. If a native method is not supported an exception is thrown. Allowing handlers should be fixed soon as it is fairly simple.

11 Structuring for research

Structuring for research and expansion is a challenge and is not totally addressed in this implementation. We have tried to make the modules as independent as possible. For example the native handling or class loading could be replaced without too much effort. The next step is to create interfaces for each of the modules so that the whole JVM could be made "plug-and-play". For most of the modules this does not pose a problem and there is very little work to be done to do this. The garbage-collector, however, poses a problem because depending on how it is implemented it needs many and various hooks into the other modules. Creating a complete set of hooks would be a large challenge. This issue has yet to be fully explored.

Much work has been done to make the implemention as obvious and accessible as possible. This is possible because speed is not really a major issue and so much of the code is implemented for clarity rather than speed.

12 Efficiency

Efficiency was not a major concern with this JVM; even the simple improvements shown in [3] were not done. Performance could be improved in several ways to increase performance. Executing the JVM with a Just In Time compilers made a noticeable difference to performance as would natively compiling the code. Fortunately none of these will affect how the code actually runs (see section 14) but will only improve speed. Many simple optimisations could have been done but were left out for clarity and because of the amount of effort they would have taken.

12.1 A faster stack

The stack is used everywhere in the JVM, nearly every instruction pushes or pops some values from the stack. A minor improvement here will make a major difference to the entire performance of the JVM. The original implementation of the stack was very object-oriented and used helper classes to implement its behaviour. Each call to pop an object from the stack led to at least four function calls being performed and several object lookups. By moving the logic into the stack class itself this number was cut in half. If the stack was non-expandable this value could be improved even further but this was not done. The performance of the JVM increased by approximately one third. In this case for very little loss of clarity (some comments were added to make up the shortfall), a major performance increase was achieved.

13 Integration within the network Simulator

14 Conclusions

Determinism One of the nicest features of this JVM is that it is completely deterministic. This holds event when debugging information is added to the output. This is not true of most JVMs as they rely on thread scheduling, timers and other features of the operating system and platform on which they run. This tends to add a small amount of non-determinism to scheduling etc. and the addition of debugging output etc. would cause similar timing problems. This is excellent for simulation and debugging because two runs of the same program with the same input will always produce exactly the same result.

Testing and Proving Proving that the JVM meets the given specification exactly is very hard. It has been shown that many of the commercial releases are not perfect. Sun do not supply a test-suite for this which is annoying. We have used the regression test-suite that is supplied with KAFFE [7] for much of our testing. This provides a good set of tests to prove that the JVM is at least passable. There are many other examples written which we have produced scripts to test. It is very useful to be able to run the program against Suns implemention. This gives a good idea of whether the program is working properly although at times the Sun implementation does not always supply the expected output.

A common problem debugging these systems is that the insertions of debugging hooks alters the behaviour of the program. Our JVM did not have this problem and we have debugging hooks set that can be switched on and off dealing with function calling, opcode execution and scheduling. This was not a complete debugging environment but it certainly helped. It might also have been useful to test parts of it while running in an IDE but this was not possible at the time.

Accurate Execution speed Currently every instruction within the JVM is assumed to take one “step”. This is not completely accurate. Some instructions take much longer to execute than others. Work needs to be done profiling other JVMs to find out accurate figures. It is quite easy to work these back into the current JVM implementation to provide an even more accurate model. This is not considered a problem and the JVM is considered accurate enough for our current needs.

Integration within network simulators JVM simulation is CPU intensive. Therefore it is difficult to run a number of simulated JVMs on a single machine. This severely limits the size of simulations and therefore its practicality. We have moved towards distributing our simulation environment across multiple machines. This is only feasible because network simulation is well suited to distribution.

Where Next? The implemention is a good clean implemention of a basic JVM. We would recommend taking another implementation and converting it to your needs, if possible, rather than starting from scratch. To truly finish this JVM there is much work that needs to be done but it has been taken to a point where it meets our needs for experimentation. This project will be publically released shortly and work will probably continue with the help of others to make it a really complete project. We will shortly be using this JVM in simulation work related to Active Networking which will show how useful it really is.

References

[1] Andrew W. Appel. *Modern compiler implementation in Java: Basic Techniques*. Cambridge University Press, 1997. Gives a useful basis for the design of VMs and a set of notes on garbage collection.

[2] Magnus Hjersing and Anders Ive. Javax - an implementation of the java virtual machine. Master's thesis, Department of Computer Science, Lund Institute of Technology, Sweden, December 1996. Implementation in C without GC.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series from the Source. Addison-Wesley, September 1996. A good specification but stops short on implementation details.

[4] Sun Microsystems Ltd. Sun java homepage http://java.sun.com/.

[5] Jon Meyer and Troy Downing. *Java Virtual Machine*. The Java Series. O'Reilly, March 1997. Concentrates too much on Jasmin, but has slightly more notes than Sun series book.

[6] Scott Oaks and Henry Wong. *Java Threads*. The Java Series. O'Rielly, January 1997. Useful for the understand of what is going on with threads.

[7] Kaffe Team. Kaffe,a free virtual machine to run java code. Available at http://www.kaffe.org/. The amount of time and effort that has obvious gone into this is amazing, my complements to those involved.

[8] NS Development Team. Lbnl network simulator ns2. Available at http://www-mash.cs.berkeley.edu/ns/. The standard network simulator for network protocol research. Written in a mixture of TCL and C++ (Strange...).

[9] Sun Java Team. Java developers kit 1.1.5. Available at http://java.sun.com. The home to java.

[10] Sun Java Team. Java developers kit 1.1.5 source release. Available at http://java.sun.com/source/. The source code for JDK1.1.5 (its big).

[11] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[12] Unknown. An implementation of the java virtual machine. Master's thesis, USA, Unknown. Implementation in C without GC.

[13] Bill Verner. *Inside the Java Virtual Machine.* McGraw-Hill, 1998. Excellent introductory book and the example applets are great.

[14] Paul R. Wilson. Uniprocessor garbage collection techniques. In *1992 International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science.