Running a Java VM Inside an Operating System Kernel *

Takashi Okumura Bruce Childers Daniel Mossé

Department of Computer Science
University of Pittsburgh
{taka,childers,mosse}@cs.pitt.edu

Abstract

Operating system extensions have been shown to be beneficial to implement custom kernel functionality. In most implementations, the extensions are made by an administrator with kernel loadable modules. An alternative approach is to provide a run-time system within the operating system itself that can execute user kernel extensions. In this paper, we describe such an approach, where a lightweight Java virtual machine is embedded within the kernel for flexible extension of kernel network I/O. For this purpose, we first implemented a compact Java Virtual Machine with a Just-In-Time compiler on the Intel IA32 instruction set architecture at the user space. Then, the virtual machine was embedded onto the FreeBSD operating system kernel. We evaluate the system to validate the model, with systematic benchmarking.

Categories and Subject Descriptors D [*4*]: 4

General Terms Design, Management

Keywords Java Virtual Machine, Just-In-Time compilation, Kernel extensibility

1. Introduction

The kernel is typically a sanctuary for code hackers and experts. Most average users are unable to modify the kernel, although there are many advantages to allowing the execution of user-level programs inside an operating system (OS) kernel. The primary advantage is the ability to extend the kernel functionality in an user-specified way. In particular, Java, due to its prominence and its write-once run-anywhere property, helps to ease the development and the deployment of kernel extensions.

To use Java for kernel extensions requires that the Java Virtual Machine (JVM) be embedded directly within the kernel. We call such a JVM an "in-kernel Java virtual machine" because the virtual machine is executed in privileged mode in the kernel address space. When the JVM is embedded in the kernel, it is important to ensure that its memory and performance overheads do not adversely impact the kernel extension, the kernel itself, or the applications running on the system. Also, some guarantee of safety has to be provided so that a kernel extension does not inadvertently corrupt the operating system, or even worse, provide an avenue for a system attack. Indeed, this virtual machine approach has been successfully taken in kernel extension studies, particularly for network I/O [16, 15, 3, 27, 4, 7, 9] and Java has been used to bring about more programming flexibility [24, 13, 14, 10, 2, 23].

While implementations of the JVM have been used for kernel extensions, they have tended to rely on bytecode interpretation. An advantage of the interpretation approach is its simplicity of the implementation. This benefit is a favorable property that assures code portability. However, an interpretation approach can harm system performance. Ahead-of-time compilation is another possibility, but it restricts the "run anywhere" advantage of using the JVM to provide portable extensions. Certain issues with safety are also more problematic with this approach. Finally, for the few in-kernel JVMs that do exist, there has been minimal discussion of the technical challenges in their implementation. For example, little has been described about how one can debug a virtual machine running in privileged mode, or how one can efficiently deliver a classfile to the in-kernel virtual machine.

In this paper, we describe our experience in developing and implementing an in-kernel Java Virtual Machine for FreeBSD running on the Intel IA32 instruction set architecture. As a case study, we focus on the flexible customization of network I/O code in the operating system by the user. Our in-kernel JVM employs just-in-time (JIT) compilation, with several lightweight code optimizations, to ensure that the user extensions execute efficiently. It also uses several simple but effective strategies to provide protection assurances when executing a kernel extension.

This paper's contributions include:

- An empirical proof that an in-kernel Java Virtual Machine does not significantly jeopardize operating system performance, contrary to the common belief.
- The work found practical optimizations to efficiently execute packet processing programs in kernel, and execution profiles to guide future optimization efforts in this domain.
- The work produced an efficient lightweight JVM for IA32, reusable for many other purposes, with a variety of *lessons learned* from our experience, which will benefit other researchers and practitioners in their implementations of an extensible kernel with a virtual machine.

This paper is organized as follows. First, a design overview of the virtual machine with its Just-In-Time compiler is presented in Section 2. This is followed by a systematic evaluation and discussion in Section 3. Section 4 describes the lessons learned in the implementation and the embedding process, along with techniques to further optimize the in-kernel execution of Java programs. We give a review of related work in Section 5, and conclude the paper in Section 6.

* Supported in part by NSF grants ANI-0325353, CNS-0524634, CNS-0720483, CNS-0551492, and CCF-0702236

VEE'08, March 5–7, 2008, Seattle, Washington, USA.

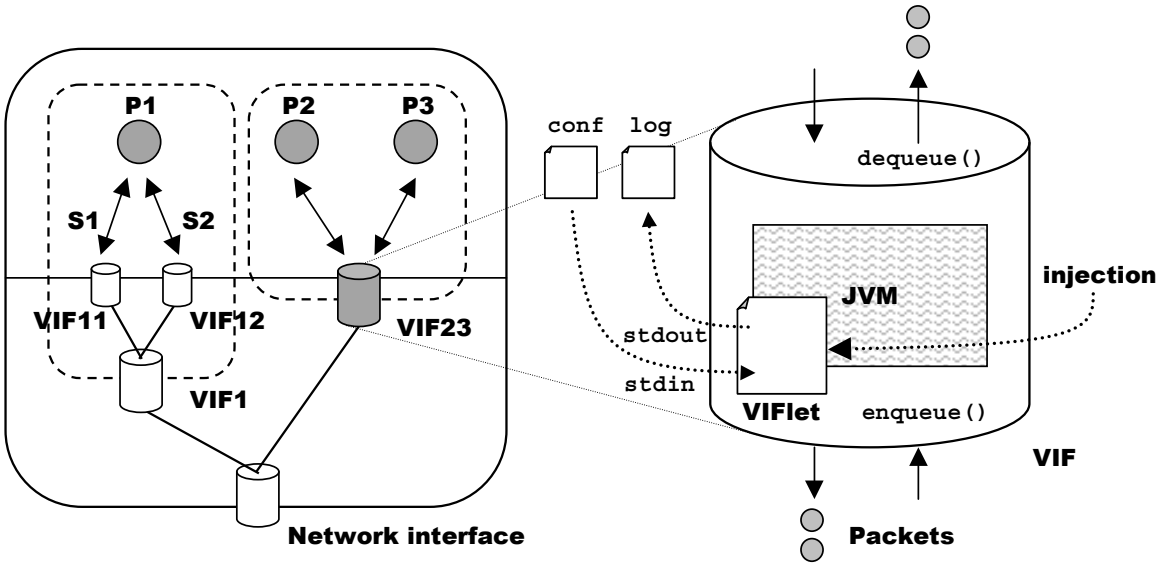

Figure 1. Overview of the in-kernel lightweight JVM

2. Design of the VM/JIT

2.1 Background and Overview

We have developed a new control model of network I/O in which an everyday user and application programs can safely control network I/O without system administrator privileges. Our approach relies on a novel virtualization model, called *hierarchical virtualization* of network interfaces [18]. The model transforms network interfaces into a hierarchical structure of virtual interfaces (VIF) and connects the VIFs into terminating entities, such as processes, threads, and sockets (Figure 1-left). This simple model accomplishes partitioning of the physical resource and provides isolation of administrative domains among users and applications at a desired granularity. It is also not limited just to a single interface or to the entire system. For example, in the figure, control made on VIF1 affects only traffic for process P1. If finer granularity of control is needed, P1 may simply use VIF11 or VIF12 to control the traffic for sockets S1 and S2, respectively, without affecting traffic for P2 and P3.

Because the virtualization model provides isolation of traffic, it is a powerful framework for allowing custom packet processing. Accordingly, we designed a programming model, called a VIFlet, which allows the injection of packet processing code into the kernel without system-wide administrator privileges (Figure 1-right). A VIFlet is a Java-based software component that includes packet processing event-handlers. A user and his/her applications can inject a VIFlet into each VIF they control to create a customized behavior for packet processing.

When a packet arrives at the VIF, the system invokes event-handlers to appropriately process packets and move the packets to the next stage of processing, namely `enqueue()` and `dequeue()` in the figure. In our VIFlet model, these are two Java methods in the `VIFlet` class. Users are allowed (encouraged) to override these methods to specify what should be done when a packet arrives (enqueue) and when a packet should be sent to the next VIF or to the user entity (dequeue). Because the VIFlets are written in Java, even a novice programmer can write his/her own packet processing code, and thus benefit from this *kernel extension* mechanism.

A sample code of the programming model is shown in Figure 2 to illustrate the actual use of VIFlets by processing packets according to some priority rules, rather than FIFO order. First,

```
public class PriQ_VIFlet extends VIFlet {
    private static final int NCLASS = 4;
    private Queue[] queue;
    private PacketClassifier pc =
                new SimplePacketClassifier();

    PriQ_VIFlet() {
        System.out.println("Initializing...");
        queue = new Queue[NCLASS];
        for (int i = 0; i < NCLASS; i++)
            queue[i] = new Queue();
    }

    public void enqueue(Packet p) {
        int type = pc.classify(p);
        length++;
        queue[type % NCLASS].enqueue(p);
    }

    public Packet dequeue() {
        for (int i = 0; i < NCLASS; i++)
            if (queue[i].isEmpty() != true) {
                length--;
                return queue[i].dequeue();
            }
        return null;
    }
}
```

Figure 2. Sample VIFlet code

we declare a VIFlet by inheriting the `VIFlet` class. The example, `PriQ_VIFlet`, is a VIFlet that does simple priority queuing. In the constructor, we initialize Queue objects, which keep actual packets. The methods `enqueue()` and `dequeue()` are event handlers for packet handling and packet processing. These are invoked when their associated events occur: packet arrival and packet drainage.

The example VIFlet classifies incoming packets through the `SimplePacketClassifier` class (not shown), and queues packets based on the packet type returned by the packet classifier. Although the VIFlet model is a simple concept, this example illustrates that the model has enough descriptive power to realize queuing controls. It is easy to see that VIFlet can also perform packet filtering, by selectively dropping packets.

To empirically verify the feasibility of the proposed scheme, and to see if Java programs can perform the packet processing task at practical performance inside the OS kernel, we implemented an in-kernel Java Virtual Machine (called NVM), with an associated just-in-time compiler (called NYA). Our prototype was implemented as an extension to the VIF framework [18], which is an open source system developed for the BSD and Linux operating systems. User interface of the system is provided through the `/proc` file system abstraction. VIFs are represented as a hierarchical directory structure under `/proc/network/`. In our prototype, users put a VIFlet (a VIFlet classfile or archived classfiles) into a VIF directory with appropriate write permissions. Once inserted in the directory, the VIF is immediately and automatically injected into the kernel, due to `/proc` and VIF structures. Output from the in-kernel JVM or VIFlet are sent to a log file in the same directory of the VIFlet. Relying on the `/proc` file system has the advantage of little overhead and all in-memory processing.

2.2 NVM: A lightweight Java VM

VIFlets are processed in the kernel via a Java Virtual Machine. To allow integration with the kernel, the virtual machine implementation needs to be kept modular and compact. NVM is one such lightweight JVM that was designed to be embedded in the OS kernel. It is based on Waba VM [25], which is a simplified Java interpreter that targets small computing devices. Waba VM's source code is open to the public and its simplified interpreter engine is highly portable to various architectures.

To serve as an in-kernel JVM, we needed to heavily modify the Waba VM for three reasons. First, the interpreter was too slow because it interprets each instruction one by one. Second, Waba requires an independent class library file. For an in-kernel VM, a more appropriate solution is to integrate the VM and the library class file to simplify the implementation. Lastly, Waba's class loader takes just a simple class file as input; it can not read a Java archive (JAR) file. To avoid excess interaction between kernel space and user space, the class loader must be more sophisticated.

For these reasons, we modified the original Waba VM and implemented various components and tools. NYA, outlined in the next subsection, is a Just-In-Time compiler designed for the virtual machine. The class library was highly tuned for in-kernel execution of packet processing code (e.g., eliminating RMI). Furthermore, the class loader was rewritten, with a decoder for JAR files and for built-in classes.

2.3 NYA: A Just-In-Time compiler for NVM

Although NVM can execute VIFlet programs as well as ordinary Java class files, as mentioned above, interpretation is not a practical solution to execute Java programs in the kernel. NYA is a Just-In-Time (JIT) compiler, developed to boost the execution performance of NVM.

Because each VM has its own memory model, a JIT must be designed for each VM and written almost from scratch. For example, the internal representation of a class greatly differs from VM to VM, and object instances are represented differently in memory. For this reason, NYA is developed specifically for NVM, reusing the bytecode management framework of TYA [12], which is an open source JIT for the Sun Classic VM on the IA32 architecture.

NYA performs simple table-driven translation of a bytecode into native instructions. Accordingly, the next challenge was how to further boost the performance with code optimization. To this end, various optimizations were added, as shown in Table 1, and described below.

- OPT_REGCACHE caches frequently used local variables on the registers

OPT_REGCACHE	Register cache of locals
OPT_COMBINEOP	Instruction folding
OPT_COMBINELD	Further instruction folding
OPT_ACACHE	Enables array object cache
OPT_OBJECT	Enables fast object access
OPT_INLINING	Inlines small functions
OPT_NULLIFY	Nullifies void functions
OPT_INLINECLK	Inlines clock syscall
OPT_FASTINV	Invokes methods fast
OPT_MCACHE	Enables method cache

Table 1. Optimization Options

- OPT_COMBINEOP performs instruction folding, which generates simpler instructions for idiomatic sets of successive bytecodes, such as arithmetic operations on the stack
- OPT_COMBINELD attempts further instruction folding on bytecode, spanning more than two instructions
- OPT_ACACHE boosts the access to array objects by simple caching
- OPT_OBJECT achieves faster access to objects by optimizing address expressions
- OPT_INLINING inlines small methods, which eliminates invocation overhead for these methods
- OPT_NULLIFY eliminates invocation of void methods
- OPT_INLINECLK is an option to boost the access to system clock, which is often used in packet processing
- OPT_FASTINV boosts method invocation; this is done because usually method invocation of an object oriented language is a complex task, which requires runtime determination of the actual callee
- OPT_MCACHE boosts the method call by simple caching

2.4 Customization for Performance

The options described above are optimizations to execute Java programs faster, in general. To further boost the execution of packet processing programs, customizations are made to the virtual machine by removing unused features and adding useful native routines. This implies that NVM supports a subset of Java – rather than the entire Java specification. Although these changes violate the standard rules for creating a JVM, our approach is similar to the one used for specialized Java execution environments. For example, Java 2 Micro Edition (J2ME) specified the K Virtual Machine, which is designed for embedded systems with limited memory and processing power [22]. The difference is that we specialized the JVM for packet processing. The changes made include:

- Because most variables for packet processing do not require 64 bits, 64-bit variables such as `long` and `double` are substituted by 32-bit `int` and `float`. This change greatly simplifies the VM design on a 32-bit architecture.
- We eliminated Java constructs that are not typically useful for packing processing with VIFlets, including `Exception`, `Thread`, and `Monitor`. The JVM simply aborts a VIFlet when it finds such operations, as a security violation. Users are expected to assure code integrity and avoid the use of these constructs.
- We removed code verification to speed up JVM processing and reduce response time. Indeed, NVM does not perform code verification before execution. Rather, we guarantee system security by aborting execution of a VIFlet when an illegal instruction is executed.

- We provided native functions for routine processing, such as checksumming. This is because it would still be costly for Java programs to intensively access the packet payload, which requires checking of the addresses legitimacy in each step of the payload access. Since we want to enhance the performance of packet processing, it is important to optimize data access to the packet payload.

These customizations not only improved the execution performance, they also had a favorable impact on the code size of the virtual machine and JIT. The resultant virtual machine (NVM) is just 64KB in size, including its class library. The size of the Sun classic JVM for the same architecture is 981KB and it also requires a standard class library file of 8907KB (JDK1.1.8). Additionally, the NYA JIT is only 64KB in size. For comparison, the size of the JIT compilers TYA and shuJIT, which are used in our performance evaluation, are 84K and 132K, respectively. Despite these changes, NVM does support standard Java mechanisms such as inheritance and interfaces. It can run ordinary Java bytecode generated by ordinary Java compilers, without any modification.

2.5 Protection mechanisms

Because VIFlets allow any user to write software code that can be executed in the kernel, the NVM/NYA needs protection mechanisms to assure the isolation of code execution. It is possible that a malicious or careless programmer could create a VIFlet with an infinite loop, where each iteration of the loop invokes many methods or does some heavy computation. To ensure resource protection, we implemented two mechanisms, in addition to the hierarchical virtualization framework for code isolation.

First, a **stack depth counter** is implemented to abort a VIFlet when the run-time stack exceeds a certain threshold. For this purpose, the system keeps a global variable which is decremented at method call, and incremented at method return. Utilizing the simple mechanism, a VIFlet that inadvertently makes too many recursive calls will be aborted, when the stack depth exceeds the threshold. Note that the JIT uses the kernel stack to hold the Java stack. This arrangement has simple and fast address calculations.

Second, a **software timer** is used to abort a VIFlet's execution when a time threshold is exceeded. To implement the timer, we instrumented the code at special locations, adding a counter (say, of number of instructions executed). The JIT emits instructions necessary to keep track of the number of backward branches taken in a method. Whenever a backward branch is taken, the counter is decremented to check if the value is still positive. When the counter reaches zero, the in-kernel JVM invokes an abort routine to terminate the execution of the program.

Because NVM/NYA does not perform code verification, a malicious user may inject Java classes with bogus bytecodes. Such bytecodes could cause the program to execute in an unexpected manner. However, the effect of this action is strictly limited to the Java heap because the system restricts accesses to stay within the Java environment. The code is gracefully terminated by the protection mechanisms when execution exceeds the given limits, or executes an unsupported bytecode, affecting solely that user's packets. An expensive alternative to this "reactive" approach is to implement a verifier that has to be used prior to running a VIFlet.

3. Evaluation and Discussion

This section presents the performance profile of the NVM and NYA implementations. We also discuss future improvements.

3.1 Basic Performance Profile

We executed several benchmarks to profile the prototype implementation. For the experiments, we utilized an experimental computing platform with the following specifications. **Hardware:** Intel Celeron 2.0Ghz CPU, Gigabyte GA-8INXP motherboard (Chipset: E7205), and 512MB of main memory. **Software:** FreeBSD 4.11 and JDK1.1.8.

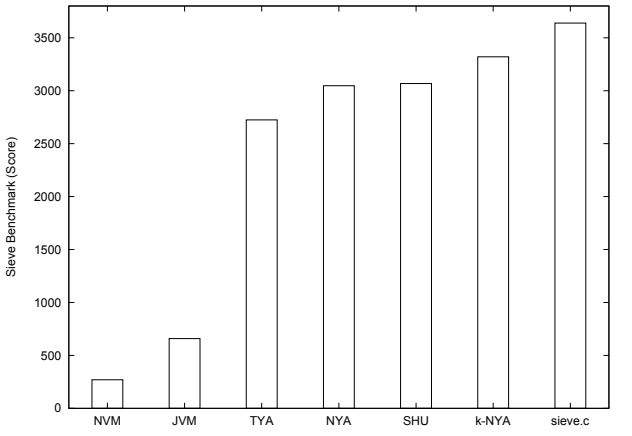

Figure 3. Sieve Benchmark

Raw Performance Benchmark We first executed the Sieve benchmark, a widely available CPU-intensive benchmark for performance evaluation of Java platforms. It is a simple benchmark that iteratively performs the Eratosthenes' prime number sieve algorithm. It shows the number of iterations in a period as a Sieve score. We used this benchmark to compare our implementation with other JVMs and JITs. The measurements in Figure 3 are taken in user space, unless otherwise noted. We did not use the timer protection feature (loop detection) because it would quickly terminate the benchmark with a timing violation.

The lowest performance was by NVM, *i.e.*, our *interpreted* virtual machine. The Sun classic JVM interpreter (labeled "JVM"), bundled with JDK1.1.8 performed 2.4 times faster than NVM. TYA and shuJIT are open source JITs for the Sun JVM, and they boosted the performance of the Sieve benchmark by five times over Sun's JVM. Our JIT, NYA, had equivalently good performance as shuJIT, exhibiting a substantial improvement over the NVM interpreter.

To investigate in-kernel execution of NVM, we measured its performance on the Sieve benchmark when hosted inside the kernel. The result is shown in the figure by the bar labeled "k-NYA". As shown, there was a 10% increase in the Sieve score, which is the best performance among all the JITs in the figure. The improvement is due to the non-preemptive property of in-kernel execution of the JVM.

Finally, we measured the performance of the Sieve benchmark as a native program written in C. The Sieve score for this program is shown by the bar labeled `sieve.c` in the figure. It is noteworthy that k-NYA is within 91% of the native C performance and is 11 times faster than the interpreter. This result shows that the virtualized code need not significantly jeopardize system performance.

Object handling benchmark The Object sort benchmark checks performance of object handling and array access by creating, sorting, and checking object arrays. The results in Figure 4 are normalized to the NVM interpreter, which had the lowest performance. The Sun JVM's performance was close to the interpreted NVM. TYA and shuJIT performed three times better than NVM. NYA with optimizations outperformed the other JITs by 100% and NVM by a factor of 6. This gain is ascribed to the optimizations applied by NYA, such as object caching.

In this study, no counterpart was provided for the native code because there are no objects in C and it is hard to make a fair

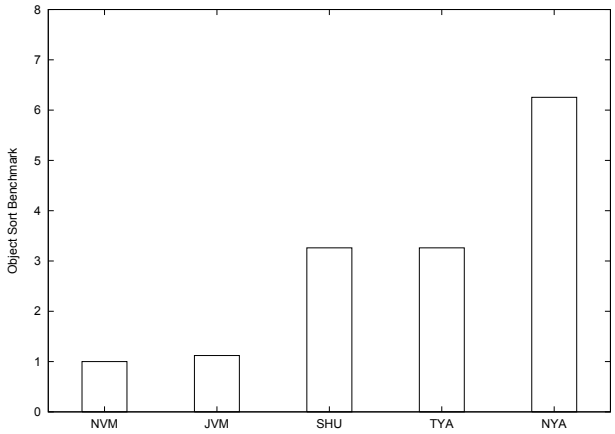

Figure 4. Object Sort Benchmark (normalized to NVM)

comparison. This situation is in contrast with the Sieve benchmark, where we can run almost identical programs in C and in Java.

Performance Breakdown In this study, we measured execution time to perform primitive operations of Java bytecode, such as method invocation, array access, ALU, object creation, etc. Because we have already described the performance of the classic JVM, we did not include it in the study to ease the readability of the graph.

Results are shown in Figure 5 – note the log scale of the Y-axis. As expected, the JITs greatly improved the performance of the interpreter, as follows.

- We note that NYA makes the execution of a loop 60 times faster, while method invocations are about 3 orders of magnitude faster with the JIT.
- We noticed that shuJIT has the best performance in the invocation benchmarks. This performance is because shuJIT has a better inlining criteria, and inlines the invoked methods more often than the other two. Note that too much inlining requires extra memory, and may sometimes impact the instruction cache.
- Native function calls ("invoke native" in the figure) took almost the same time for the JITs; note that the numbers include execution time of the called method, not only the time to call and return from the function.
- All implementations showed similar performance in the ALU benchmarks.
- Field operations (*i.e.,* reads and writes on class fields and object fields), exhibited irregular performance: shuJIT again outperformed NYA and TYA, except for static field accesses, where all JITs had similar performance.

A difference appeared in the access of local variables, where NYA did the best. Interestingly, from the results of the Object sort benchmark, we predicted that NYA would have the best performance in array access, which turned out not to be true. On the other hand, NYA created new objects and arrays with the lowest cost.

This result suggests that the code would be fastest if one (i) avoids method invocation and carefully designs the in-lining criteria, (ii) reduces the native call cost, and (iii) avoids packet object creation in the kernel. Because packet processing is a hot-spot of network systems (*i.e.*, all incoming and outgoing packets are processed), it is best to pre-allocate necessary objects at system boot time.

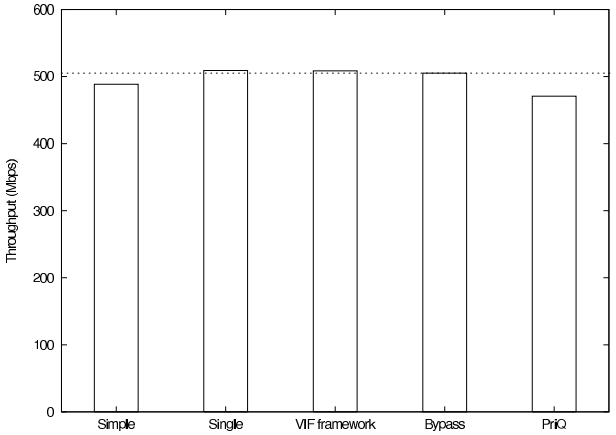

Figure 6. VIFlet Performance (Throughput)

Benchmarking a VIFlet To measure performance of the VIFlet system, we first measured the throughput of a simple VIFlet, which just relays packets. For this purpose, we generated a TCP stream from a VIFlet machine to another machine, measuring the throughput at the receiving end. On the VIFlet machine, we configured it so that the measured traffic goes through just one VIF, virtualizing the GbE interface. We ran the traffic generator for 10 seconds, and used the best 1 second interval as a peak throughput for further analysis.

The results are shown in Figure 6, where "simple" refers to a VIFlet that intercepts and forwards (no extra processing) both outgoing and incoming packets, "single" is a VIFlet that controls only outgoing traffic, "VIF framework" denotes throughput of the system without running any VIFlet (*i.e.*, just running the VIF system), "Bypass" is when the VIF system is bypassed, and "PriQ" is a VIFlet that performs priority queuing.

From Figure 6 we observe the following for each VIFlet. **Simple:** the VIFlet framework has very little overhead, just a few percent overhead of the maximum throughput. **Single:** there is a slight performance increase, compared with Simple, by omitting the methods for incoming packets; this is because latency for the packet processing was bounding TCP. **VIF framework:** has almost no performance penalty. **PriQ:** just a 7% overhead suggests that a virtualized code can be a practical solution.

3.2 Compilation cost

We also measured the compilation cost of NYA to study the impact of the JIT translation. The overall compilation time (measured with the instruction `rdtsc`, which reads the CPU cycle counter) took about 150 μsec for HelloWorld, 340 μsec for the Sieve benchmark, and 260 μsec for the priority queue VIFlet (Table 2). Because the JIT compiler works in the kernel, we should avoid blocking other system functions and/or user-level programs. We investigated the cost in more detail below.

Table 2 shows that the compilation cost greatly varies from method to method and that the cost is not strictly proportional to the size of the code (time / instructions). For example, in the PriQ_VIFlet case, the compilation of $\langle init \rangle$ is more costly than other methods (enqueue and dequeue) in the class, although the instruction counts of methods do not differ so much (26-29). Actually, the first method of each class always has the highest cost per generated instruction (approximately a factor of two to three). This is because the compilation of the first method intensively calls the class loader to load basic classes, such as java.lang.String, java.lang.StringBuffer, and java.lang.System, which inflates the compilation cost of the first method.

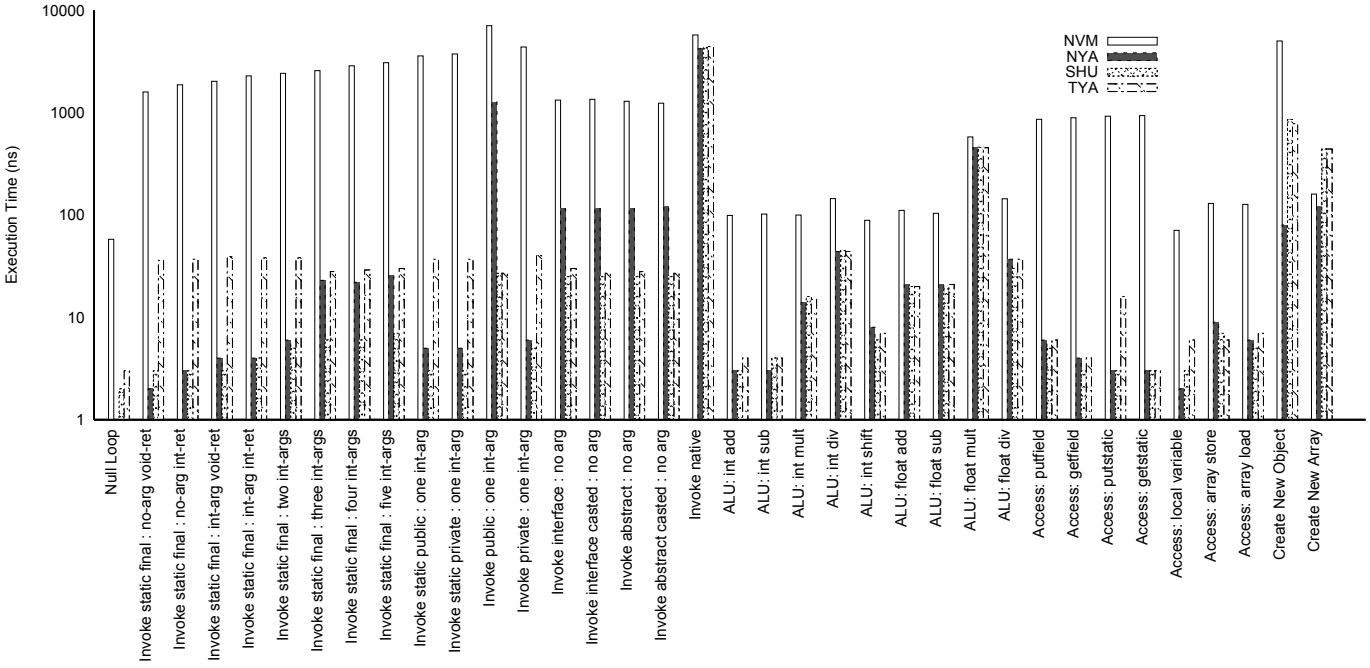

Figure 5. Java Benchmark for different operations — Note the log scale of the Y axis

Class	Method	#inst'n	time (μsec)	time / inst'n
HelloWorld	main	4	117.8	29.5
HelloWorld	$\langle init \rangle$	3	35.5	11.8
Sieve	main	14	139.0	9.9
Sieve	runSieve	109	191.0	1.8
Sieve	$\langle init \rangle$	3	11.1	3.7
PriQ_VIFlet	$\langle init \rangle$	29	131.7	4.5
PriQ_VIFlet	enqueue	26	52.3	2.0
PriQ_VIFlet	dequeue	28	79.4	2.8

Table 2. Compilation cost sample

We also studied the cost of different optimization options. Because certain processing like *inlining* clearly increases the compilation cost of a method, we expected that we could reduce the cost by turning off the options. However, we did not observe the expected gain. This was because the compilation time of the first method dominated the overall cost. Nevertheless, significant improvements in the compilation time per instruction of the other methods were observed by turning off the optimizations.

The results suggest that it would be reasonable to focus on the *class loading cost* of the first method for further performance tuning. This startup overhead was studied in [11], where it was suggested to keep an image of a memory area loaded with classes and to clone the stored image whenever needed. Another approach to address the class loader problem is to modify the class loader so as to pre-load some basic classes (java.*) embedded in the VM just once at the system boot time and use the memory in a read-only manner.

Note that, to avoid one-shot compilation at code injection, which can possibly stop the entire system for a temporary time period. We used a lazy compilation approach where compilation of each method is deferred until it is first called, rather than when its class is loaded.

3.3 Discussion

The known downside of adding a virtualization layer is that it inevitably causes overhead, as illustrated by the results described above (*e.g.*, about 7% of performance for a priority queue VIFlet). There are several approaches to tackle the overheads in the future, as discussed next.

It is costly to aggressively optimize Java bytecode on IA32 architecture inside the kernel because data flow analysis is needed for good register allocation. We could achieve better performance gain through an instruction set architecture (ISA) that is more suitable for dynamic code generation than Java bytecode, as explored in recent projects, such as LLVA [1], Dis [26], and Virtual Processor [20]. Additionally, it would be reasonable to incorporate different technologies for specific subsystems. For example, packet classification can be far more efficient with a special code generator, like DPF [7].

In fact, one may argue that there is a fair chance of gaining better performance, even with the additional overheads caused by the virtualization, when dealing specifically with network I/O subsystems. This is because unbinding of operating system kernels and protocol stacks increases optimization opportunities. Examples of such optimization include Integrated Layer Processing [6, 5], which tried to fuse the layers, like loop-fusion in compiler optimization, and specialization of the protocol stack for a particular application protocol [17]. OS extensibility also favors speculative replies to requests by in-kernel caching for networked servers. These optimizations are not widely in use, mostly because network I/O code is hardcoded inside the OS and it has been too costly to replace the working implementations only for these specific purposes.

In summary, we need to consider two aspects in the packet processing system. One is the generated code, which can be improved with an ISA suitable for dynamic code generation. The other aspect is flexibility of the code execution environment. If the VM can perform protocol processing, it allows aggressive customization of the protocol stack. If it allows more flexibility in the programs, it

may realize speculative execution and customized caching even in the kernel, which would result in further performance gain.

4. Lessons Learned in Implementing NVM/NYA

In the following subsections, we describe the implementation process of our in-kernel virtual machine (Figure 7). Indeed, the development process of an in-kernel virtual machine has rarely been reported. Accordingly, we briefly summarize the coding process, emphasizing the lessons learned. First, the class library and target VIFlets are implemented, before developing the virtual machine (Class Coding). Second, actual virtual machine was implemented (VM Implementation). Third, the virtual machine is embedded inside an operating system (Embedding). These subsections are followed by a discussion about the performance enhancement and the garbage collection inside operating system kernels.

4.1 Step 1: Class Coding

We started the implementation by building the environment to develop the virtual machine. For example, to code the virtual machine, we first needed target VIFlet programs, to test the functionality of the virtual machine being built. The target VIFlets necessitate a class library that contains the `Packet` and the `Queue` classes. To facilitate the coding process, we also needed a testing driver.

For these reasons, we first designed the Java classes, needed for VIFlet programs. Then, various VIFlets were designed, along with the actual implementation of the class library. When these classes were done, we implemented a Java tool, `VIFletTester`, which tests basic functionality of target VIFlets and the class library on IBM's Eclipse. We also created a built-in traffic generator in the `VIFletTester` program, to test the VIFlets using synthetic traffic. Then, for more realistic tests, a driver was written, which emulates packet flow, out of a dump file of Berkeley Packet Filter (BPF) [15]. Utilizing this workbench, we could debug the VIFlet classes and the class library, on top of the running JDK.

4.2 Step 2: VM Implementation

Second, after creating the necessary classes, we proceeded with the implementation of the virtual machine. To simplify the debugging of our lightweight JVM/JIT, we first coded and debugged the VM outside the kernel, in user space. For this purpose, we wrote another program, `VIFletd`. `VIFletd` is a daemon program that takes a VIFlet as an input, and executes the VIFlet program, by invoking necessary methods, such as $\langle init \rangle$ on the VM core being built.

Another helper program is the *ROMizer*. It creates a ROM image file, as a C source program, out of Java class files. The program realizes a JVM with a built-in class library, which is a preferable form of an in-kernel virtual machine, for its relative simplicity in the embedding phase. It also facilitates development and extension of the VIFlet functionality, by allowing easy modification of the class library embedded in the virtual machine.

Note that there are three sources of packet data (as shown in Figure 7) in the `VIFletd`: a device file of BPF, a diverting socket of the Netnice Packet Filter (NPF) [19] and a BPF dump file. The BPF device file is a read-only file, which injects the packet data into the `VIFletd`, capturing the live traffic. The NPF diverting socket diverts packet flow into the userland. Utilizing the packet diverting mechanism, users can actually execute the VIFlets on actual packet data path, executing a VIFlet as an application in user space. The last is a dump of BPF as mentioned above.

After implementing the VM interpreter (NVM), we debugged the VM code, with the help of the programs and the drivers explained above. Once the interpreter became stable, we implemented the JIT compiler (NYA). The most time-consuming task in the process was to program code emission routines that emit native binary code corresponding to each Java bytecode. Once the simple translation table is completed, optimization algorithms were implemented to further boost the performance, with the help of benchmarking programs described in Section 3.

4.3 Step 3: Embedding

After developing NVM/NYA in user space, the developed virtual machine is integrated into the VIF framework in the kernel [18]. Towards that goal, we embedded appropriate hooks in the VIF functions so that packets are diverted, upon arrival of packets, to the VM core for processing. After finishing the necessary hooks in the kernel, we found some additional work was needed to make the virtual machine work inside the operating system.

First, we needed to adjust the function interfaces. There are many standard functions, such as I/O functions like `read()` and `write()`, which are commonly used in ordinary programs, but not supported in the kernel. In a virtual machine, they are used in the class loader modules. Likewise, there are many functions which have different argument set in the kernel, such as `malloc()` and `free()`. These functions can be well substituted by short macros, or wrapper functions. There is another set of functions, which were thought to be unsupported, but work well even inside the kernel. For example, abort processing is frequently used in a VM code, and is implemented with `setjmp()` and `longjmp()`, which worked well even inside the kernel.

Another issue was the protection. Just-In-Time compilers sometimes utilize hardware protection mechanisms, such as "division by zero" and "page fault", to simplify runtime checks. If any of these violations happen, an exception is raised, and the operating system takes back the control. However, inside the kernel, use of the hardware protection compromises modularity of the virtual machine, because these mechanisms require hooks to the VM code in the hardware interrupt handler. Because such a hook would complicate the system design, we opted for doing runtime checks in the VIFlet itself: we substituted the runtime checks by inserting extra instructions into the code generated by the JIT.

4.4 Runtime checking and garbage collection

During the implementation, we found an important issue about the reference to packet objects in the heap. If a packet is no longer referenced, it is good practice to discard the packet as quickly as possible, not to exhaust packet buffer pool on the system. However, checking whether packets are referenced can be expensive. Suppose that a careless programmer enqueues a given packet into two queues by mistake. This operation results in duplicate references from multiple queues, and the system consistency can be compromised if the packet is actually dequeued. The straightforward solution is to sweep the heap, not to leave a bogus reference, when the packet is first dequeued. But, this is obviously impractical. Accordingly, we propose not to allow such a duplication of packet reference in a VIFlet. Such a checking is made at runtime relatively easily, by maintaining a reference counter on each packet object. The system simply aborts the execution when duplication is detected. This is another form of optimization for packet processing code, and we believe that this is not a serious restriction in packet processing programming.

Another concern to run a JVM in kernel is the garbage collector (GC), because execution of GCs freezes the VM for a certain period of time. It would severely impact the system performance, since the VM runs non-preemptively inside the kernel. However, detailed code review suggested that actual packet handlers rarely allocate memory dynamically, to avoid the performance impact. Therefore, we decided to discourage users from using `new` in the handlers, making it the user's responsibility to write programs that preallocate necessary memory in the initialization phase. In this

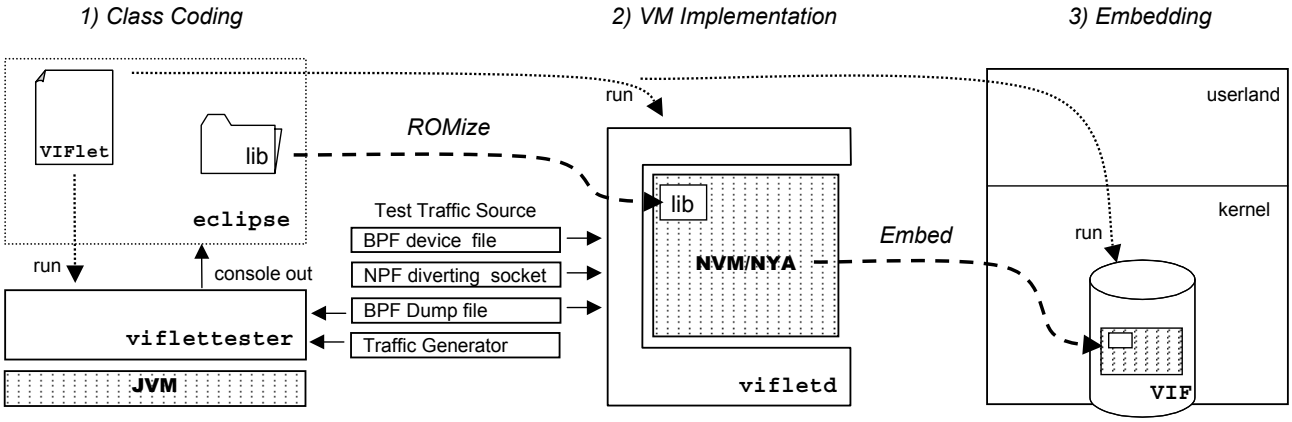

Figure 7. Overview of the coding process

fashion, GCs will not be invoked on-line. We again believe this is not a severe restriction for packet processing systems.

5. Comparison with Related Work

There are two schools of thought in the Java Virtual Machine implementations. One is the JVMs targeting toward better performance in the generic case, strictly complying with the Java specification. The other is a class of JVMs for specific purposes, which sometimes omit certain features, mostly for resource-constrained settings, such as embedded processors and cell phones. The K Virtual Machine, released by Sun Microsystems, is a representative VM in the latter class [22], which is designed for the Java 2 Micro Edition (J2ME). This Java specification is for embedded systems with limited memory and processing power. Examples include devices like Personal Digital Assistants (PDAs), and there are commercial VMs, such as Jeode EVM [21], as well as open source VMs, such as Waba [25]. These specialized Java VMs are typically designed to be compact, and forgo optimizations because the optimizer may impact their resource requirements. However, some recent work (Hotpath) showed that a trace-based dynamic optimizer can be used to efficiently transform and improve application performance, even in a resource constrained setting [8].

The Java programming language and run-time environment has also been used for coding network I/O extensions. For example, [10, 13, 14] proposed to use Java for network I/O programming, with the aim to provide code portability and a rapid prototyping framework. Other work also embedded the Java execution environment in the kernel for kernel extensions. For example, U-net/SLE [24] proposed to embed a Java interpreter into the kernel so that users can inject new interrupt handlers into a device driver. JEM tried to integrate the Java execution environment and a microkernel to allow execution of Java programs in the network layer [2].

Our work differs from these past works in several ways. One essential difference between the various approaches for in-kernel JVMs lies in the control model. All the existing proposals work either on actual individual interfaces or on all the network traffic. Because of this property, almost all the proposals have been protected by administrator privilege. We proposed a model which allows flexible extension of the kernel functionality by users, or even by untrusted applications, through resource isolation, provided by the hierarchical virtualization of network interface [18].

We differ also in the implementation approach. Most of the past approaches used a Java interpreter [24] or an Ahead-of-Time (AOT) compiler [23]. The interpreter approach falls short of the required level of performance. AOT approaches have a performance advantage over interpreter or current JIT based schemes, because AOT can afford costly compiler optimizations. Their disadvantage is their static nature; they can perform only static optimizations. On the other hand, dynamic code generation schemes can exploit runtime profile of the executing programs to further optimize the code, as is intensively studied in the research field [1, 20, 26]. Our in-kernel JVM study could lay the groundwork toward application of dynamic code generation for network I/O in the OS field.

6. Concluding Remark

There are potential advantages in allowing the execution of Java programs inside operating system kernels, as a mechanism to flexibly extend the kernel functionality. This paper described how a Java VM can be integrated into the kernel for network extensions, and made the following contributions.

First, it produced an empirical proof that virtualized code does not significantly jeopardize operating system performance, dismissing the common belief that in-kernel execution of Java program would have adverse impact on the system performance.

Second, the work found practical optimizations to efficiently execute packet processing programs in kernel, such as restriction of the Java features, restriction of packet duplication, and heuristics in garbage collection. Further, the execution profiles and the experiences of the implementation would guide future optimization efforts in the domain, for example, to reduce the class loading cost.

Third, the work produced an efficient lightweight JVM for IA32, reusable for many other purposes, with a variety of *lessons learned* in the implementation process of in-kernel virtual machine. They would be useful to other researchers and practitioners in implementing their own kernel extension mechanism with the virtual machine technology.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. Llva: A low-level virtual instruction set architecture. In *International Symposium on Microarchitecture*, pages 205 – 216. IEEE/ACM, December 2003.

[2] M. Baker and H. Ong. A java embedded micro-kernel infrastructure. In *Java Grande*, page 224, 2002.

[3] A. Begel, S. McCanne, and S. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.

[4] H. Bos and G. Portokalidis. Fairly fast packet filters. http://ffpf.sourceforge.net/.

[5] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap - role-based architecture. In *First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[6] T. Braun. Limitations and implementation experiences of integrated layer processing. In *GI Jahrestagung*, pages 149–156, 1995.

[7] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.

[8] A. Gal, M. Franz, and C. W. Probst. Hotpathvm: An effective JIT for resource-constrained devices. In *2nd International Conference on Virtual Execution Environments (VEE 2006)*, jun 2006.

[9] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. Keromytis. xpf: packet filtering for lowcost network monitoring. In *HPSR2002 (the IEEE Workshop on High-Performance Switching and Routing)*, pages 121–126, May 2002.

[10] M. Jung, E. Biersack, and A. Pilger. Implementing network protocols in java - a framework for rapid prototyping. In *International Conference on Enterprise Information Systems*, pages 649–656, 1999.

[11] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable jvm: A new approach to start isolated java applications faster. In *3rd International Conference on Virtual Execution Environments (VEE 2007)*, jun 2007.

[12] A. Kleine. TYA. http://www.sax.de/~adlibit/.

[13] B. Krupczak, M. H. Ammar, and K. L. Calvert. Implementing protocols in java: The price of portability. In *INFOCOM (2)*, pages 765–773, 1998.

[14] B. Krupczak, K. L. Calvert, and M. H. Ammar. Increasing the portability and re-usability of protocol code. *IEEE/ACM Trans. Netw.*, 5(4):445–459, 1997.

[15] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[16] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[17] G. Muller, R. Marlet, C. Pu, and A. Goel. Fast, optimized sun rpc using automatic program specialization. In *The 18th International Conference on Distributed Computing Systems*, pages 240–249, May 1998.

[18] T. Okumura and D. Mossé. Virtualizing network I/O on end-host operating system: Operating system support for network control and resource protection. *IEEE Transactions on Computers*, 53(10):1303–1316, 2004.

[19] T. Okumura and D. Mossé. Netnice packet filter - bridging the structural mismatches in end-host network control and monitoring. In *INFOCOM2005*. IEEE, March 2005.

[20] I. Piumarta. The virtual processor: Fast, architecture-neutral dynamic code generation. In *Virtual Machine Research and Technology Symposium 2004*, pages 97–110, May 2004.

[21] Insignia Solutions. The jeode platform : Accelerated java solutions for internet appliances. White Paper, May 1999.

[22] Sun Microsystems, Inc. KVM white paper. May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[23] M. Welsh. Safe and efficient hardware specialization of java applications. Technical report, University of California, Berkeley, May 2000.

[24] M. Welsh, D. Oppenheimer, and D. Culler. U-net/sle: A java-based user-customizable virtual network interface. *Scientific Programming*, 7(2):147–156, 1999.

[25] R. Wild. Waba. http://www.wabasoft.com.

[26] P. Winterbottom and R. Pike. The design of the inferno virtual machine. In *IEEE Compcon 97*, pages 241–244, February 1997.

[27] M. Yuhara, B. Bershad, C. Maeda, and J. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *the USENIX Winter 1994 Technical Conference*, pages 153–165, January 1994.