User Customization of Virtual Network Interfaces with U-Net/SLE

David Oppenheimer and Matt Welsh
{davidopp,mdw}@cs.berkeley.edu

December 9, 1997

Abstract

We describe U-Net/SLE (Safe Language Extensions), a user-level network interface architecture which enables per-application customization of communication semantics through downloading of *user extension applets*, implemented as Java classfiles, into the network interface. This architecture permits applications to safely specify code to be executed within the NI on message transmission and reception. By leveraging the existing U-Net model, applications may implement protocol code at the user level, within the NI, or using some combination of the two. Our current implementation, using the Myricom Myrinet interface and a small Java Virtual Machine subset, obtains good performance, allowing host communication overhead to be reduced and improving the overlap of communication and computation during protocol processing.

1 Introduction

Recent work in high-speed interconnects for distributed and parallel computing architectures, particularly workstation clusters, has focused on development of network interfaces enabling low-latency and high-bandwidth communication. Often, these systems bypass the operating system kernel to achieve high performance; however the features and functionality provided by these different systems vary widely. Several systems, such as U-Net [26] and Active Messages [27], virtualize the network interface to provide multiple applications on the same host with direct, protected network access. Other systems, including Fast Messages [16] and BIP [17], eschew sharing the network in lieu of design simplicity and high performance. In addition, fast network interfaces often differ with respect to the communication semantics they provide, ranging from "raw" access (U-Net) to token-based flow-control (Fast Messages) to a full-featured RPC mechanism (Active Messages). Complicating matters further is the spectrum of network adapter hardware upon which these systems are built, ranging from simple, fast NICs which require host intervention to multiplex the hardware [30] to advanced NICs incorporating a programmable co-processor [26].

Application programmers are faced with a wide range of functionality choices given the many fast networking layers currently available: some applications may be able to take advantage of, say, the flow-control strategy implemented in Berkeley Active Messages, while others (such as continuous media applications) may wish to implement their own communication semantics entirely at user level. Additionally, the jury is still out on where certain features (such as flow-control and retransmission) are best implemented. It is tempting to base design choices on the results of microbenchmarks, such as user-to-user round-trip latency, but recent studies [12] has hinted that other factors, such as host overhead, are far more important in determining application-level performance.

Given the myriad of potential application needs, it may seem attractive to design for the lowest common denominator of network interface options, namely, the interface which provides *only* fast protected access to the network without implementing other features, such as RPC or flow-control, below the user level (U-Net is one such design). This design enables applications to implement protocols entirely at user level and does not restrict communication semantics to some arbitrary set of "built-in" features. However, experience has shown [10] that in the interest of reducing host overhead, interrupts, and I/O bus transfers, it may be beneficial to perform some protocol processing

[0] For more information on this project, please see `http://www.cs.berkeley.edu/~mdw/projects/unet-sle/`.

within the network interface itself, for example on a dedicated network co-processor [5]. Such a system could be used to implement a multicast tree directly on the NI, allowing data to be retransmitted down branches of the tree without intervention of the user application, overheads for I/O bus transfer, and process or thread context switch. Another potential application is packet-specified receive buffers, in which the header of an incoming packet contains the buffer address in which the payload should be stored. Being able to determine the packet destination buffer address before any I/O DMA occurs enables true zero-copy as long as the sender is trusted to specify buffer addresses.

A number of systems have incorporated these NI-side features in an *ad hoc* manner, however, it would seem desirable to have a consistent and universal model for fast network access which subsumes all of these features. We have designed an implemented U-Net/SLE (Safe Language Extensions), a system which couples the U-Net user-level network interface architecture with user extensibility by allowing the user to download customized packet-processing code, in the form of Java applets, into the NI. With this design, it is possible for multiple user applications to independently customize their interface with the U-Net architecture without compromising protection or performance. Applications which are content with the standard model provided by U-Net are able to use "direct" U-Net access and are not penalized for features provided by the underlying system which they do not use.

With the U-Net/SLE model, for example, it is possible for an application to implement specialized flow-control and retransmission code as a Java applet which is executed on the network interface. For instance, the semantics of the Active Messages layer could be implemented as a combination of Java and user-level library code. Those applications which require the use of Active Messages may use those features without requiring all applications on the same host to go through this interface, while still being able to take advantage of the ability to do NI-level processing rather than pushing all protocol code to user level.

A number of questions are raised by U-Net/SLE and other similar designs, including:

- How can performance be maintained while protection between user extensions is enforced?
- How should resource management for user extensions be handled?
- At what point does it become advantageous to move packet-processing code from user level into the NI, and vice versa?
- What tradeoffs should be considered when providing a certain feature in the network interface rather than on the host?

In this paper we describe the design and implementation of U-Net/SLE, focusing on the considerations of using Java as a safe language for user extensions executed in the critical path of communication on a network interface.

Section 2 of this paper describes the U-Net/SLE design in more detail. Section 3 describes JIVE, our implementation of the Java Virtual Machine used in U-Net/SLE. Sections 4 and 5 summarize performance, while section 6 describes related work. Section 7 concludes and raises issues for future work to consider.

2 Design and Implementation

U-Net/SLE is based on U-Net [26], a user-level network interface architecture which multiplexes the NI hardware between multiple applications such that each application has transparent, direct, protected access to the network. U-Net may be implemented either in hardware, software, or a combination of both, and does not presume any particular NIC design. On NICs with a programmable co-processor, for instance, U-Net multiplexing/demultiplexing functions may be implemented directly on the co-processor, while on a non-programmable NIC a protected co-routine on the host may be used to enforce protection.

In the U-Net model an *endpoint* serves as an application's interface to the network and consists of a *buffer area* and *transmit*, *receive*, and *free buffer queues* (see Figure 1). The buffer area is a pinned region of physical host RAM mapped into the user's address space; in order to ensure that the NI may perform network buffer DMA at any time,

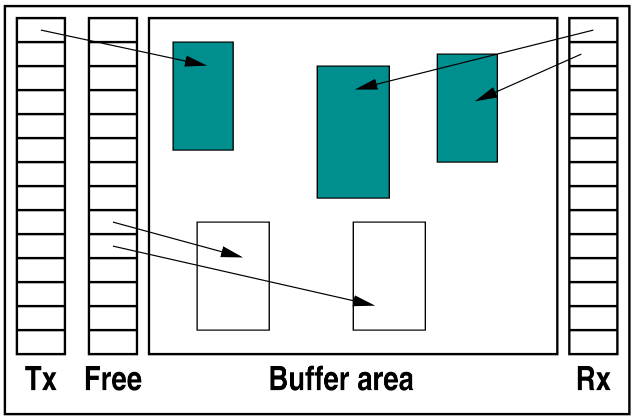

Figure 1: U-Net endpoint data structure

all transmit and receive buffers are located in this region.[1] In order to transmit data, the user constructs the data in the buffer area and pushes a descriptor on the transmit queue indicating the location and size of the data to be transmitted as well as a *channel tag*, which indicates the intended recipient of the data. U-Net transmits the data and sets a flag in the transmit queue entry when complete. When data arrives from the network, U-Net determines the recipient endpoint for the packet, pops a buffer address from that endpoint's free buffer queue and transfers the data into the buffer. Once the entire PDU has arrived (which may span multiple receive buffers), U-Net pushes a descriptor onto the user receive queue indicating the size, buffer address(es), and source channel tag of the data. As an optimization, small messages may fit entirely within a receive queue descriptor. The user may poll the receive queue or register an upcall (*e.g.* a signal handler) to be invoked when new data arrives.

U-Net/SLE allows each user endpoint to be associated with a Java classfile implementing the *user extension applet* for that endpoint.[2] This applet consists of a single class which must implement the following methods:

- `UnetSLEApplet`, the constructor for the class;
- `doTx`, invoked when the user pushes an entry into the transmit queue; and,
- `doRx`, invoked when a packet arrives for the given endpoint.

In addition the class must contain the field `RxBuf`, an unallocated array of `byte`. This array is initialized by U-Net/SLE to point to a temporary buffer used for data reception; see below. The class need not adhere to any additional structural restrictions. The use of Java as a safe user extension language is discussed in section 3.

When an endpoint is created, if the user has supplied a classfile it is loaded into the network interface, parsed, and initialized by executing the applet constructor.[3] This constructor could, for example, allocate array storage for future outgoing network packets, or initialize counters.

During normal operation, U-Net/SLE polls the transmit queues for each user endpoint while simultaneously checking for incoming data from the network. When a user pushes a transmit descriptor onto their transmit queue, U-Net/SLE first checks if the endpoint has a classfile associated with it. If not, U-Net/SLE transmits the data as usual. Otherwise, the applet `doTx` method is invoked with three arguments: the length of the data to be transmitted, the destination channel, and the offset into the user's buffer area at which the payload resides. These arguments correspond exactly to the contents of the transmit descriptor. The `doTx` method may then inspect and modify the

[1] The U-Net/MM architecture [32] extends this model to permit arbitrary virtual-memory buffers to be used; the future work section of this paper discusses it further.

[2] This terminology used because, like standard Java applets, U-Net/SLE applets are mini-programs that register callback functions with the runtime environment. The use of the term *applet* is not meant to suggest that U-Net/SLE applets are required to import the `java.applet` Java package or that they are otherwise constrained by the restrictions on standard Java applets.

[3] The current prototype provides a single application classfile buffer, which is loaded when the network interface is initialized.

packet contents before transmitting it to the network (if at all). Multiple packets may be injected into the network as a result of the `doTx` call.

Similarly, when data arrives from the network, U-Net/SLE first determines the destination endpoint. If this endpoint has a classfile associated with it, the applet's `doRx` method is invoked with the packet length and source channel as arguments. The packet at this point resides in the applet's `RxBuf` buffer, although implementations may choose not to implement this feature (if, for example, a direct network-to-host DMA engine is available). This method may process the packet contents, allocate and fill user free buffers, and may eventually push one or more receive descriptors into the user's receive queue. If no applet is associated with this endpoint the data is pushed to the user application as described above.

Operations such as moving data between user and network interface buffers, and pushing data to or pulling data from the physical network, are handled by *native methods* provided to user extension applets by U-Net/SLE. These methods are responsible for enforcing protection boundaries between user applets; for example, to prevent an applet from accessing data outside of the user endpoint buffer area. The exact functionality provided by these native methods is dependent on the U-Net/SLE implementation; for example, on some NICs it may be necessary to move data into a temporary buffer before the applet can process it. In general, however, these methods must ensure that applets may only access (a) the user transmit, free, and receive queues, and (b) the user buffer area. The methods need not protect the applet from writing "garbage" data into these regions; any such bug is considered a user programming error and will not affect other applications.

2.1 Myrinet interface implementation

Our prototype implementation uses the Myricom Myrinet SBus interface, which incorporates 256K of SRAM and a 37 MHz programmable processor (the LanAI), with a raw link speed of 160 MBytes/sec. The host is a 167 MHz UltraSPARC workstation running Solaris 2.6. U-Net/SLE is implemented directly on the LanAI processor, with the raw U-Net functionality being very similar to that for the FORE Systems SBA-200/PCA-200 implementations described in [26, 31]. Endpoint transmit and free queues are mapped into user space from the LanAI SRAM, while pinned host memory is used for the endpoint receive queue and buffer area. This allows the LanAI to poll endpoint transmit queues and users to poll their individual receive queues without crossing the I/O bus. The Myrinet board interface requires network data to be transmitted from and received into buffers in the LanAI SRAM; there is no fly-by DMA capability such as that described in [26]. However, the network-to-SRAM and SRAM-to-host DMA engines may operate in parallel.

U-Net/SLE is implemented as an add-on to the standard U-Net firmware running on the LanAI, and user extension applets are executed on the LanAI itself in response to user transmit requests and data reception from the network. At endpoint creation time a user application may load the unparsed Java classfile for their applet into a region of the LanAI SRAM, where it is then parsed and the applet constructor executed. The Java virtual machine implementation, JIVE, is described section 3.

The following native methods are provided by U-Net/SLE for applets to process packet data:

`doHtoLDMA(byte[] txarray, int txoff, int useroff, int len)` DMAs a region of host memory specified by *useroff* (an offset into the endpoint buffer region) into the Java array *txarray* at offset *txoff* for *len* bytes.

`doLtoHDMA(byte[] rxarray, int rxoff, int useroff, int len)` DMAs a region of LanAI memory specified by the Java array *rxarray* starting at offset *rxoff* into the endpoint buffer region at offset *rxoff* for *len* bytes.

`txPush(byte[] txarray, int txoff, int chan, int len)` Transmits a packet to destination channel *chan*, using the payload stored in *txarray* starting at offset *txoff* with size *len* bytes.

`dmaRxd(int[] offsets, int chan, int len)` DMAs a descriptor into the user's receive FIFO specifying channel tag *chan* and length *len* with receive buffer offsets *offsets*. The *offsets* array must be equal to the size of the receive descriptor offset list (fourteen 32-bit words in this implementation).

`dmaRxdPayload(byte[] payload, int chan, int len)` DMAs a descriptor into the user's receive FIFO specifying channel tag *chan* and length *len* with receive descriptor payload specified by the *payload* array. This allows small message reception to be optimized by placing message data directly in the receive queue entry.

`getFreeBuffer()` Returns the buffer area offset of the next free buffer on the user's free queue, or -1 if the queue is empty.

`getFreeBufferLength()` Returns the size of this endpoint's free buffers, which is specified at endpoint creation time.

`newAlignedArray(int nbytes)` Returns a `byte` array of size *nbytes* which is properly aligned for DMA operations.

Some of these native methods are appropriate for all implementations of U-Net/SLE (such as `txPush` and `dmaRxd`), while others (such as `doHtoLDMA`) are specific to the Myrinet implementation. No restrictions are made as to where these native methods may be invoked, or even in what order; the applet `doTx` method could, for example, allocate and fill a user receive buffer.

```
public class RawUnet {
  public static byte[] TxBuf;
  public static byte[] RxBuf;
  public static int freebufferlen;
  public static int[] offsets;

  RawUnet() {
    TxBuf = newAlignedArray(4096);
    freebufferlen = getFreeBufferLength();
    offsets = new int[14];
  }

  private static int doTx(int length, int channel, int off) {
    doHtoLDMA(TxBuf, 0, off, length);
    txPush(TxBuf, 0, channel, length);
    return 0;
  }

  private static int doRx(int length, int channel) {
    if (length <= 56) {
      dmaRxdPayload(RxBuf, channel, length);
    } else {
      int dma_offset = getFreeBuffer();
      if (dma_offset == -1) return; /* No buffer available; drop */
      doLtoHDMA(RxBuf, 0, dma_offset, length);
      offsets[0] = dma_offset;
      dmaRxd(offsets, channel, length);
    }
    return 0;
  }
}
```

Figure 2: Sample U-Net/SLE applet source code

Figure 2 shows sample U-Net/SLE applet code that implements the standard U-Net mechanism; that is, it simply transmits and receives data without modifying it. For simplicity the applet assumes that a single receive buffer will be sufficient to hold incoming data. A more complicated applet could modify the packet contents before transmission, or generate acknowledgment messages for flow-control in the receive processing code. Section 5 evaluates the performance of several applets.

3 Java Virtual Machine Implementation

In this section we discuss the design and implementation of the Java virtual machine subset used in U-Net/SLE, Java Implementation for Vertical Extensions (or JIVE). An implementation of U-Net/SLE need not use JIVE as its virtual machine; JIVE is simply one virtual machine with sufficient functionality to perform the operations needed by many useful U-Net/SLE applications yet small enough to run on a network interface.

3.1 JIVE design

JIVE implements a subset of the Java Virtual Machine [11] and executes on the LanAI processor of the Myrinet network interface. Our goals in designing JIVE were simplicity, a small runtime memory footprint, and reasonable execution speed even on a relatively slow processor. All three goals stem from characteristics common in an embedded processor environment like that of the LanAI: a limited runtime system, limited memory resources, and a CPU slower than that found in workstations of the same generation.

JIVE defines a subset of the Java virtual machine, not the Java language [7]. The design of JIVE therefore does not force any choice of source language from which the virtual machine bytecodes are compiled. In practice, however, we expect that Java will be the language used for programming JIVE. Because JIVE implements a subset of the Java VM, JIVE classfiles can be generated by a standard Java compiler.

We compare JIVE to the standard Java VM in three areas: type-related features, class- and object-related features, and runtime features.

JIVE supports the `byte`, `short`, and `int` datatypes, and one-dimensional arrays of those datatypes. JIVE does not support the `char`, `double`, `float`, or `long` datatypes, or multi-dimensional arrays. We feel that this latter set of datatypes is unlikely to be needed by an applet that performs simple packet processing, which is the design target for JIVE. Although trivial to support, the `char` datatype has been excluded from JIVE because the Java VM's requirement that `char`'s be represented as 16-bit unsigned integers representing Unicode version 1.1.5 characters [25]. A packet processing applet wishing to read or write the contents of a packet would use the `byte` datatype, which is 8 bits and therefore matches the size of the C `char` datatype; it is difficult to envision a network protocol whose packet meta-information (*e.g.* headers) would be encoded in Unicode.

A JIVE applet consists of a single class. Because a JIVE applet consists of a single class, JIVE need not support non-array objects except for a single instance of the applet's class. Array objects are supported, and arrays are treated as objects (*e.g.* it is legal to invoke the `arraylength` operation on an array reference). Dynamic class loading is not necessary because a class is associated with an endpoint at the time the endpoint is instantiated. Arbitrary user-defined methods are fully supported. Methods and class variables may be static or non-static; since only one instance of the applet class will ever exist at a time, the semantics of static and non-static functions and class variables are identical.

JIVE does not support interfaces, exceptions, threads, or method overloading. These features would increase the runtime overhead and code size of JIVE and are unlikely to be useful to packet processing applets. The current prototype implementation of JIVE does not support garbage collection; the addition of a garbage collection or a scope-based persistence model is a potential area of future work. JIVE supports the invocation of native methods exported by the U-Net/SLE LanAI Control Program that allow applets to perform operations such as DMA to and from a user-level application and transmitting data to the network. Exceptions are a useful feature that should be added in a future implementation of JIVE; they were left out of the current version due to time constraints.

The current implementation of JIVE assumes a trusted Java compiler. Bytecode verification could be incorporated into a trusted host daemon that is invoked when a JIVE classfile is loaded into the network interface, thus removing this assumption. There is little reason to make bytecode verification part of the JIVE virtual machine executing on the network interface: since bytecode verification only needs to be performed once each time a class is loaded, overlapping bytecode verification with other work on the host unlikely to improve overall system performance. In addition to the standard bytecode verification for safety, a bytecode verifier for JIVE should also ensure that the classfile being loaded conforms to the subset of the Java VM that JIVE supports.

3.2 Java as an extension language

We selected Java as the user extension language for U-Net/SLE for a number of reasons:

- Safety. Java's safety features mesh well with the U-Net model of protected user-level access to the network interface. An unsafe language without some external safety mechanism (such as Software Fault Isolation [29] or Proof-Carrying Code [13]) requires blindly trusting the compiler that generated the code (as in SPINE [5], which requires the compiler to sign downloaded code). The Java sandbox, as enforced by the bytecode verifier and runtime checks, protects applets from one another. Combined with resource consumption limits enforced by the runtime system, these safety features provide a strong guarantee that a user cannot interfere with other users by writing a malicious applet.
- Speed. Java bytecode can be interpreted or compiled to native machine code. As we will see in section [section], the latter approach offers performance very close to that of native machine code, while still providing Java's safety guarantees.
- Compact program representation. Java class files are very compact. Many operations take their operands from, and push their result to, the stack, and can therefore be encoded in a single byte (compared to four bytes to encode the native instructions of most modern workstations and embedded controllers) because the source and destination are implicit. For some operations that use local variables, the Java Virtual Machine defines special bytecodes that explicitly encode the local variable to be used (for the most common local variables); this allows operations that would otherwise be encoded in two bytes (a one-byte opcode followed by a one-byte local variable number) to be encoded in a single byte, in the common cases. For example, the bytecode `ISTORE_3` instructs the virtual machine to pop one value off the stack and to store it in local variable 3.
- Portability. Because Java bytecodes are platform-independent, a JIVE applet can be written and compiled once, and then run on any network interface with a JIVE virtual machine implementation. This makes it easy to implement a network protocol that runs on a heterogeneous collection of hosts and network interfaces. For example, a Java classfile could be sent as part of a network packet in an active network [23] and could run on any network interface or router with a JIVE runtime system.
- Development environment. A number of high-quality Java development environments are currently available, making development of Java code relatively easy on almost any platform. Moreover, Java is gaining popularity as an embedded programming language, so we expect a proliferation of development tools targeted to the needs of embedded systems (*e.g.* highly optimizing Java compilers, Java bytecode compression systems, and so on).

On the other hand, certain Java features are unnecessary for our purposes:

- Object orientation. The most useful object concept for a packet processing application is the idea of a packet or message as an object, with associated methods triggered between message arrival in the network interface and transfer to the application, and between transfer from the application and transfer to the network. A natural way to integrate this idea with U-Net is to consider each message/packet associated with a single endpoint to be an object of the class associated with that endpoint. Arbitrary user-defined classes beyond the single class per U-Net endpoint are not necessary in this model.
- Threads. Network packet processing is an event-driven operation: packets arrive at the network interface from the network and from the application. These events happen on a time-scale of microseconds so the amount of time needed to handle each event must be minimized. The overhead of generating a new thread to handle each such event and incorporating a thread scheduler into the Java runtime system is likely to have a detrimental effect on performance, so a threaded model seems inappropriate for packet processing handlers running on network interfaces. On the other hand, a timer mechanism could be useful, as it would allow handlers to block briefly and timeout if the event they are waiting for does not happen soon enough.

The Java Virtual Machine does not define a model for resource accounting or limitation. Resource limitation is important for U-Net/SLE, in order to guarantee that a malicious packet handler running on behalf of one U-Net endpoint cannot significantly affect the performance of a packet handler running on behalf of another U-Net endpoint. Some resource limits can be enforced at class loading time, while others can only be enforced at runtime. The resource limitations that can be enforced at load time include classfile code size and memory consumed by local variables and the operand stack in each method. The resources that can only be limited only at runtime include CPU time consumption, dynamic memory allocation due to arrays, and memory needed to store local variables and the operand stack due to recursive function calls.

Classes that exceed these resource limitations could be dealt with in a number of ways. The class loader could refuse to load classes that violate the static load-time constraints. The virtual machine could kill an applet that exceeds runtime memory constraints, possibly sending a notification to the associated user-level application or, if exceptions are supported, could throw a virtual machine exception to the applet. The virtual machine could do the same for an applet that has consumed too much CPU time. In either case the applet exception handler would be expected to rectify the situation (by reducing memory or CPU time consumption) within a fixed period of time; failure to comply would result in the applet being killed.

3.3 JIVE implementation

As mentioned earlier, JIVE aims for a small code size and small runtime memory overhead. In the first respect, the JIVE library for the LanAI is only 43K compiled, representing about 2700 lines of C source code. In contrast, Kaffe [33], a free Java Virtual Machine that implements most of the Java Virtual Machine specification, is about 15,000 lines of C source code, even when the code for just-in-time compilation and garbage collection is removed. Also, JIVE assumes no runtime library (*e.g.* no *libc*): functions for operations such as memory allocation and string manipulation are an explicit part of the JIVE library.

Function calls and operations on class fields are among the most expensive operations in JIVE. These two types of operations are time-consuming because of the way Java bytecodes refer to them in the instruction stream. Part of the Java classfile is the *constant pool* which contains, among other things, what amount to pointers to pointers to the names of class fields and methods (which are themselves also stored in the constant pool). A field/method is referenced in the instruction stream as an index into the constant pool. This index is turned into a pointer to the field/method name, and the pointer must be compared to the pointers to the fields/method names stored in the virtual machine data structures representing the classes/fields themselves, to find the proper field/method itself. [4] In the case of method lookup there exists the possibility of multiple methods with the same name (*i.e.* in the case of method overloading), in which case the formal and actual argument types must be compared to identify the correct method. By prohibiting method overloading JIVE eliminates the need to deal with that possibility, improving the performance of method invocation. Also, since an applet contains a single class, JIVE does not need to determine whether a field or method referenced in the instruction stream is defined in the current class or in another class, nor does it need to check the "access flags" on fields and methods – qualifiers like "private" and "public" are meaningless in the context of a single class. These simplifications serve to reduce the complexity and size of the virtual machine implementation and to speed up method and field access.

JIVE for the LanAI supports native methods exported by U-Net/SLE as described in section 2. The call path from the virtual machine to the native method function itself is optimized for fast invocation: since the locations for the code for the native methods are known when JIVE is first invoked, these function pointers can be stored directly in the method structures representing each of the native methods in the internal virtual machine representations for those methods.

One shortcoming of the existing JIVE prototype is that it does not enforce resource limitations on applets (such as CPU time or memory consumption). Rather than implement an arbitrary resource management policy, we believe future research is needed to find the best mechanism for resource sharing among multiple user extension applets.

[4] Kaffe compares the actual names rather than the pointers. This is likely a major contributor to the performance superiority of method invocation in JIVE compared to Kaffe.

4 JIVE Performance

We evaluated the performance of JIVE on a Sun Ultra-1 workstation running Solaris 2.5.1. The Ultra-1 contains a 167 MHz UltraSPARC I CPU with a 16K on-chip I-cache and a 16K on-chip D-cache. Our evaluation consisted of microbenchmarks, designed to ascertain the performance of JIVE on operations we expect to be commonly used by packet processing applications, and macrobenchmarks, designed to give a rough idea of JIVE's performance on "real" applications.

Java bytecode can be interpreted or can be compiled to machine code when a class is loaded. This latter method of execution is often referred to as *just-in-time* compilation because the compilation is deferred until the first time a class is loaded[5]. A virtual machine that uses interpretation will generally execute a number of native machine instructions for each bytecode, while a virtual machine that uses compilation will execute close to one native machine instruction for each bytecode (not counting the instructions needed to compile the bytecode to native machine code). The current version of JIVE uses interpretation; a later version may use just-in-time compilation. In the current version of JIVE, most bytecode operations are executed using fewer than fifteen native SPARC machine instructions. A few operations, such as those that operate on arrays, those that duplicate multiple operand stack entries, those that operate on class fields, and those involved in method calls, translate into substantially more instructions: about 25, 35, 40, and 75, respectively. The actual number of instructions required to perform method calls and operations on class fields varies because the location where a method or field mentioned in the instruction stream is stored must be looked up in a table; we have given a rough estimate of the fewest instructions that would be executed.

We compared JIVE to

- Optimized C code (compiled with `gcc`)
- Unoptimized C code (compiled with `gcc -O3`)
- Kaffe version 0.9.2 using just-in-time compilation
- Sun's Java Virtual Machine version 1.1.5
- Kaffe version 0.9.2 using interpretation

The above configurations are listed in order of decreasing expected performance.

The comparison between the Java virtual machines and C code was made by executing a C program equivalent to the Java program being tested. The execution time measurements of JIVE and Kaffe excluded class loading and initialization; it was not possible to exclude class loading and initialization time from the Sun JVM measurements because we do not have the source code for that system. In general we ran enough iterations of the Java benchmarks to ensure that class loading and initialization time would contribute insignificantly to overall execution time.

The performance metric we used for the microbenchmarks was time per measured operation, *e.g.* the amount of time needed to write one value to an array. The performance metric used for the macrobenchmarks was time per program execution. These numbers were obtained by running each benchmark repeatedly in a loop, dividing the total time consumed by the number of iterations, and subtracting out the loop overhead (*i.e.* the time needed to increment the loop counter, test for loop termination, and branch back to the beginning of the loop).

4.1 Microbenchmarks

We measured the performance of nine microbenchmarks:

- *null-loop*: loop N times doing nothing; the time per iteration for this benchmark is subtracted from the runtime of the other benchmarks

[5] Just-in-time compilation may compile individual methods instead of an entire class, in which case compilation is deferred until a particular method is invoked.

- *null-call*: call a method that takes void and returns one int
- *field-read*: read a static class field N times
- *field-write*: write a static class field N times
- *array-read*: read the same array element N times
- *array-write*: write the same array element N times
- *math*: compute $x+x+x+x+x+x+x+x+x+x$ and store it in local variable y
- *math-field*: same as *math* but x and y are static field variables of the class rather than local variables
- *math-array*: same as *math* but x and y are one-element arrays

We also measured the time for a native method call in JIVE.

The results, presented in terms of microseconds per operation, are presented below. We do not report the time for optimized C because in some cases the microbenchmark was optimized out of the executable entirely when optimization was enabled, making the measured operation appear to take "zero" time. In general our microbenchmarks were so simple that significant optimization over the "standard" gcc-generated code was not possible.

Microbenchmark performance, in microseconds per operation

benchmark	JIVE	Unoptimized C	Kaffe (just-in-time)	Sun JVM	Kaffe (interpreted)
null-loop	0.75	0.08	0.09	0.41	0.95
null-call[6]	2.32	0.12	0.28	1.07	9.47
field-read	1.38	N/A	0.13	0.57	2.02
field-write	1.41	N/A	0.12	0.58	2.03
array-read	1.46	0.11	0.16	0.76	1.8
array-write	1.51	0.11	0.15	0.76	2.2
math	4.07	0.20	0.17	2.03	4.86
math-field	7.55	0.20	0.47	2.28	13.6
math-array	8.35	0.32	0.62	4.23	10.0

Figure 3 shows the ratio of these execution times to the execution time for unoptimized C.

The above data shows that JIVE outperforms *kaffe-interp* on all the microbenchmarks. JIVE outperforms *kaffe-interp* significantly on *null-call* and *math-field* because of its optimized function call and class field access procedures. The simplifications stem from JIVE's restrictions that applets contain only one class and that method overloading may not be used, as described in section 3. We also note that just-in-time compilation brings performance to within a factor of three of unoptimized C.

4.2 Macrobenchmarks

We measured the performance of three macrobenchmarks:

- *qsort*: Quicksort an array of N numbers which are initially sorted in reverse order. This benchmark stresses method call/return (due to its recursive structure) and array operations.
- *rand*: generate N random numbers using an additive congruential random number generator [18] with a fixed seed. This benchmark stresses the mathematical operations (particularly the more time-consuming ones like multiply, divide, and modulo) and array operations.

[6] A null native method call in JIVE takes 1.60 μsec.

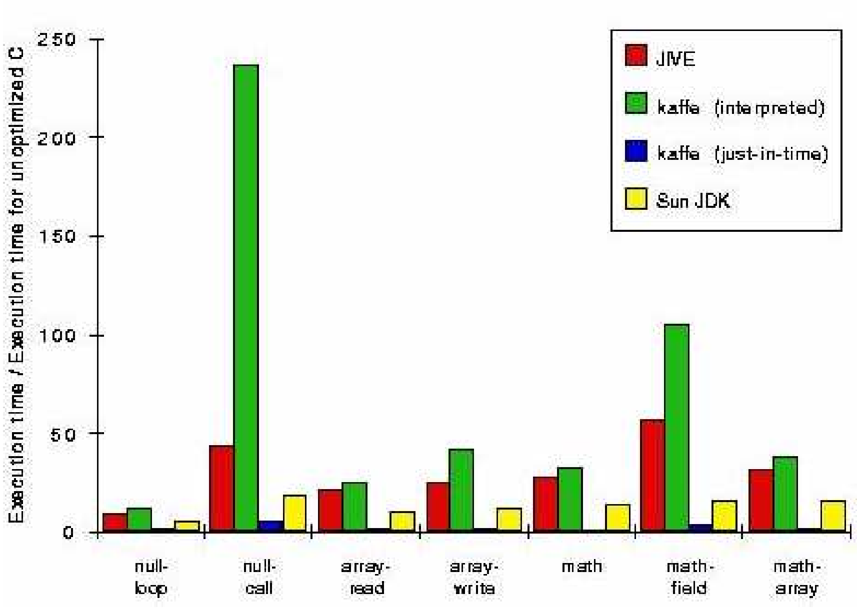

Figure 3: Microbenchmark performance

- *fibonacci*: generate the first N fibonacci numbers. This benchmark stresses the simple mathematical operations (primarily addition).

The results, presented in terms of microseconds per benchmark execution, are presented below.

Macrobenchmark performance, in microseconds per execution

benchmark	JIVE	Optimized C	Unoptimized C	Kaffe (just-in-time)	Sun JVM	Kaffe (interpreted)
qsort	24800	520	2130	2750	11700	46800
rand	11.8	1.1	3.8	3.2	5.5	11.0
fibonacci	2.15	0.018	0.19	0.16	1.04	2.52

Figure 4 shows the ratio of these execution times to the execution time for unoptimized C.

This data demonstrates that JIVE outperforms interpreted kaffe on two of the three benchmarks, and performs within 8% on the remaining benchmark. The performance of JIVE is comparable to that of interpreted kaffe on the *rand* benchmark because in executing *rand* the interpreter performs many long-latency multiply, divide, and modulo operations, the execution time for which is bound by the latency of the CPU functional unit performing the operation. In other words, optimizations in the interpreter aren't likely to lead to substantial performance improvements since a large portion of the execution time is fixed by the hardware. JIVE significantly outperforms kaffe-interp on the *qsort* benchmark, most likely because its simpler (and therefore faster) method lookup procedure reduces the overhead of function calls, as discussed earlier. This performance difference indicates that substantial performance benefit can be obtained by allowing only a single class and forbidding function overloading.

As was seen with the microbenchmarks, just-in-time compilation boosts performance substantially: with just-in-time compilation, Kaffe performed *better* than unoptimized C for two of the three benchmarks, and performed less than ten times slower than optimized C in all cases.

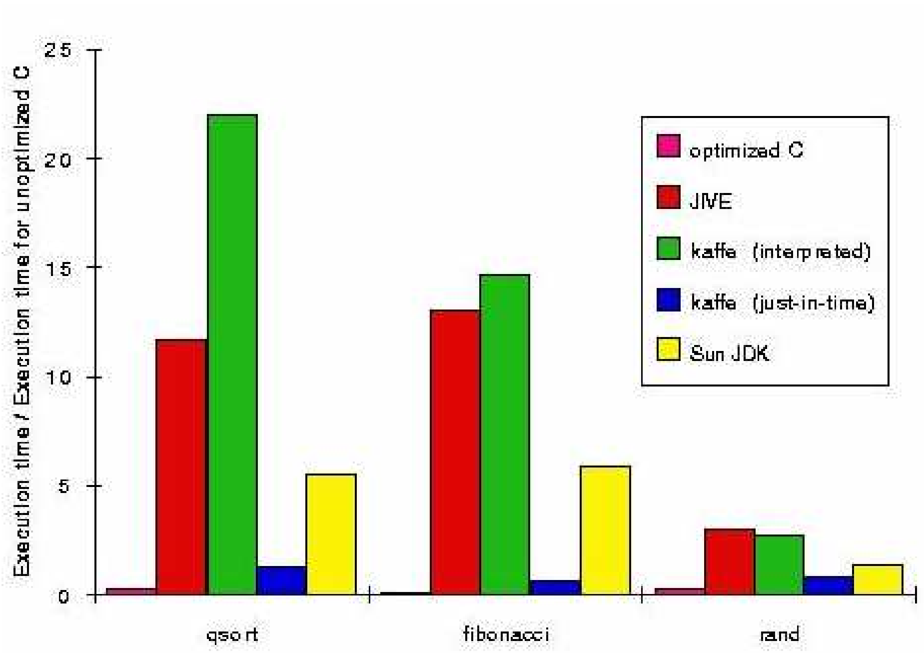

Figure 4: Macrobenchmark performance

5 U-Net/SLE Performance

In this section we discuss the performance of U-Net/SLE with JIVE for various micro-benchmarks and a variety of user extension applets running on the LanAI processor.

5.1 Latency and bandwidth measurements

Figure 5 shows round-trip latency as a function of message size for four configurations: A standard applet implementing basic U-Net semantics; a simplified applet assuming that packets will consume a single receive buffer (shown in Figure 2); an applet which performs pingpong operations between user extension applets only, without propagating messages to user level; and standard U-Net without the use of SLE. The standard U-Net applet adds between 41.2 μsec (for small messages) and 99.7 μsec (for large messages) of overhead in each direction, while the simplified U-Net applet reduces large-message overhead to 42.5 μsec. These overheads are detailed in the next section. As can be seen, round-trip latency between Java applets alone is very low, ranging between 64 and 119 μsec. This suggests that synchronization between user extension applets on different NIs can be done very rapidly.

Figure 6 shows bandwidth as a function of message size for a simple benchmark which transmits bursts of up to 25 messages of the given size before receiving an acknowledgment from the receiver. This is meant to simulate a simple token-based flow-control scheme. The applets demonstrated include the standard U-Net applet; an applet which implements the receiver-acknowledgment between applets only (without notifying the user process); an applet which transmits a burst of 25 messages for each transmit request posted by the user; and one which does not perform transmit-side DMA, meant to simulate data being generated by the applet itself.

There is a notable drop in bandwidth due to the higher overhead of DMA-setup and packet processing code as implemented in Java; however, we believe that user applications which are able to utilize the programmability of the network interface to implement more interesting protocols will be able to avoid worst-case scenarios such as those shown here. For instance, the SLE applet which implements token-based flow-control relieves the programmer from dealing with this issue at user level, allowing the application to treat U-Net/SLE as providing reliable transmission (a feature not provided by the standard U-Net model). In this way an application will be able to asynchronously

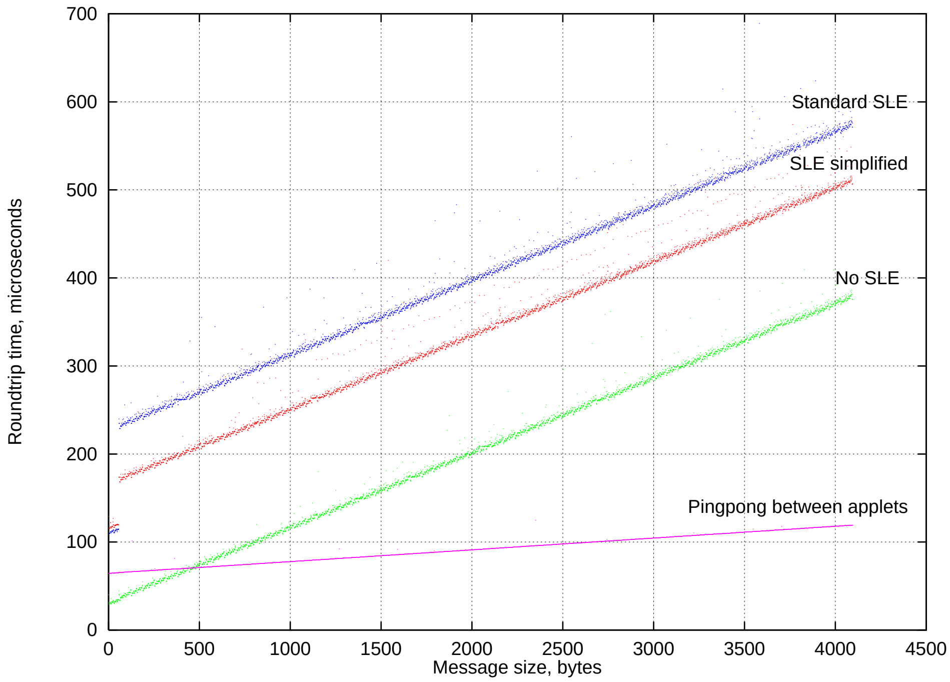

Figure 5: Round-trip latency vs. message size

receive data into its buffer area while performing other computation; no application intervention is necessary to keep the pipeline full. It should also be noted that applications are not required to use SLE features for all communication, and may wish to transmit high-bandwidth data through the standard U-Net interface while utilizing SLE extensions for other protocol-processing code.

5.2 U-Net/SLE overhead breakdown

Figure 7 shows the breakdown of overheads for various U-Net/SLE operations as executed on the LanAI processor. Note that these times do not include, for instance, DMA transfer and packet transmission times; instead they measure only the overheads for these operations, as executed through JIVE and the U-Net/SLE native methods, over the standard U-Net code. The transmit overhead regardless of message size is 24.5 μsec, while receive overhead is 16.7 μsec for messages 56 bytes or smaller, and 42.5 μsec for messages larger than 56 bytes.

As can be seen, there is an overhead of 3.2 to 3.9 μsec to invoke a user applet method from the U-Net firmware, and 5.5 μsec for null native-method invocation from JIVE. Considering the complexity of the code and the flexibility afforded by Java, these overheads are well within reasonable bounds.

The overhead for Java operations can be attributed partly to the fact that JIVE interprets Java bytecodes (rather than using just-in-time compilation), and the relatively slow clock speed of the LanAI processor compared to most modern CPUs. The results in the previous section suggest that applying JIT to U-Net/SLE should result in significant performance advantages. Likewise we believe these results should encourage designers of high-performance network interfaces to consider higher clock speeds for the network co-processor; chips such as the DEC StrongARM run at 200 MHz and are intended for such embedded applications. While there is some amount of software optimization possible in our design, we believe that NIC-side processing can benefit greatly from higher performance NIC designs, allowing more complex processing tasks to be executed on the network co-processor.

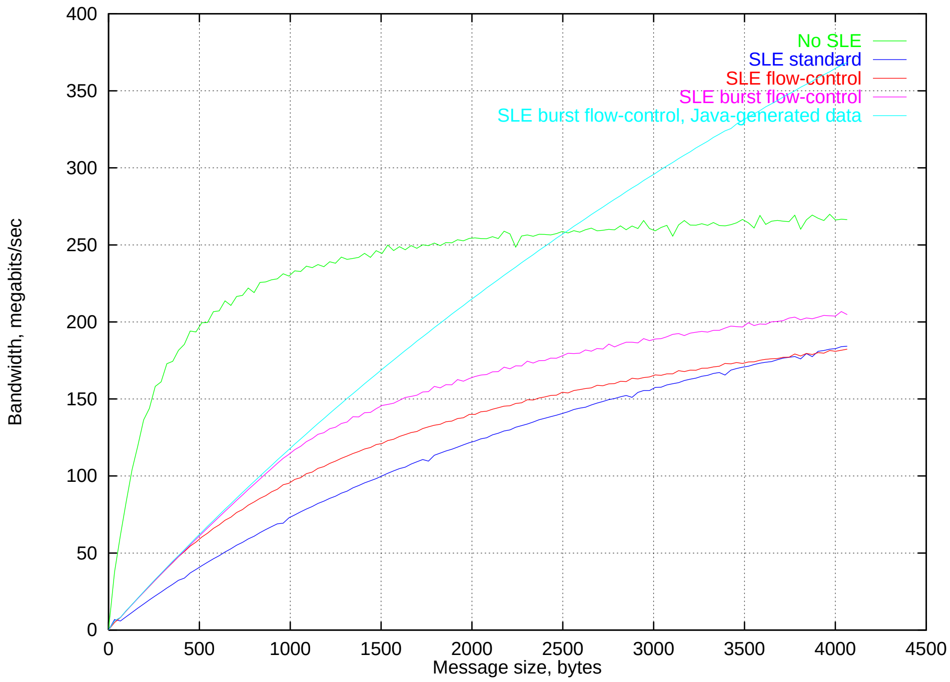

Figure 6: Bandwidth vs. message size

6 Related Work

U-Net/SLE draws on past work in the areas of programmable I/O controllers, user-supplied protocol handlers, and safe languages.

6.1 Programmable I/O controllers

One of the earliest examples of a programmable I/O controller was the I/O control units of the IBM System/360 [1]. These processors served as the interface between an I/O device's controller and the CPU. They operated independently of the CPU and could access main memory directly. The Peripheral Control Processors (PPU's) of the CDC 6600 [24] were based on a similar idea. The CDC6600 contained ten PPU's, each of which operated independently. Each PPU contained an accumulator register, could store 4K 12-bit words, and could perform 36 instructions, including mathematical, logical, and branch operations, loading/storing to/from memory, interrupting the main CPU, and transferring data to/from any of 12 "peripheral channels." The programs running on the PPU's were loaded by the system operator, but this architecture and that of the IBM System/360 represent early systems with support for programmable I/O processors.

U-Net/SLE takes advantage of "intelligent" network interfaces by downloading packet processing code to the NI much as the IBM System/360 and CDC 6600 took advantage of "intelligent" I/O controllers by dynamically loading the control programs for I/O devices. Unlike the early programmable I/O controllers described above, however, U-Net/SLE allows multiple applications to simultaneously use the network interface without interfering with one another. The operating system is not involved in providing this protection as it was in these early systems. The new technologies which enable these U-Net/SLE features are network interfaces accessible at user level and safe languages.

Transmit overhead:	**40-byte message**	**1000-byte message**
Check for classfile, setup applet call	0.7 μsec	0.7 μsec
Call applet `doTx` method and return	3.9 μsec	3.9 μsec
(Null native method call)	*(5.5 μsec)*	*(5.5 μsec)*
DMA setup overhead	11.8 μsec	11.8 μsec
Transmit data overhead	8.1 μsec	8.1 μsec
Total	**24.5 μsec**	**24.5 μsec**
Receive overhead:	**40-byte message**	**1000-byte message**
Check for classfile, setup applet call	0.6 μsec	0.6 μsec
Call applet `doRx` method and return	3.2 μsec	3.2 μsec
(Null native method call)	*(5.5 μsec)*	*(5.5 μsec)*
Get free buffer		5.85 μsec
DMA setup overhead		11.8 μsec
Do Rx descriptor DMA	12.9 μsec	21.0 μsec
Total	**16.7 μsec**	**42.5 μsec**

Figure 7: U-Net/SLE transmit/receive operation overhead

6.2 User-supplied protocol handlers

A number of systems have recently been developed that allow users to supply their own protocol handlers in place of a generic operating system handler. All of these systems, like U-Net/SLE, allow users to write special-purpose network protocols that take advantage of application-specific information and can adapt.

Application Specific Handlers (ASHs) [28] are user-supplied functions that are downloaded into the operating system[7] and are invoked when a message arrives from the network. ASHs can transfer a message from the kernel to user space, initiate an outgoing message in response, and perform general computation. ASHs operate within the address context of their associated application. The kernel ensures the safety of these programs through static and dynamic checks on the handler. The execution time of ASHs are estimated by the ASH runtime system when the ASH is loaded; any handler that exceeds its estimated runtime is killed. Wild jumps and wild writes are prevented using the host's memory protection hardware and Software Fault Isolation [29].

U-Net/SLE differs from ASHs in several respects. First, ASHs operate in kernel or user space, while U-Net/SLE extensions run within the context of the network interface (which may be embodied on a smart network co-processor or the host, or some combination of the two). Second, ASHs are triggered only when a message is received, while U-Net/SLE extensions are triggered both on receive and transmit. This second difference limits the range of uses for ASHs compared to U-Net/SLE extensions: for example, ASHs cannot turn a single user-level message send into a packet transmission to many hosts (*i.e.* a multicast) while U-Net/SLE can. Third, both U-Net/SLE and ASH allow integrated layer processing, but ASH uses dynamic code generation to compose, on-the-fly, functions that operate on network streams.[8] ASH also allows dynamic protocol composition; however, this functionality is not particularly relevant to the uses envisioned for U-Net/SLE, in which applications use single, tightly-integrated protocols rather than multiple protocols of a protocol stack.

SPIN [4] also allows users to download code into the kernel. SPIN's networking architecture, *Plexus*, runs user protocol code within the kernel in an interrupt handler. Extensions are written in the type-safe Modula-3 [14] language, and the compiler that generates the extensions is trusted to generate non-malicious code. Plexus extensions can be installed on both the transmit and receive paths; they are invoked by the operating system. Plexus extensions are based on a protocol graph which describes the protocol structure for all applications running on SPIN; an individual application installs its handlers as nodes in the graph, with safe multiplexing/demultiplexing provided by *guard* functions.

[7] The authors state that ASHs may be run in kernel or user space, but most of their discussion relates to the uses of these handlers when they are downloaded into the kernel.

[8] U-Net/SLE could achieve similar functionality by adding very clever just-in-time compilation and allowing dynamic loading of multiple classes by an applet.

SPINE [5] extends the the ideas of SPIN to the network interface, and is the system most similar to U-Net/SLE in design and scope. Underlying SPINE is a Modula-3 runtime executing on the NI, the current prototype implementation of which uses the Myrinet interface. Despite some similarities, SPINE differs from U-Net/SLE in several ways. First, SPINE targets server applications (*e.g.* it does not include fine-grain parallel communication as part of its design goals), while U-Net/SLE targets both server and cluster applications. Second, SPINE requires a trusted compiler, while U-Net/SLE takes advantage of the safety features of Java. Third, SPINE extensions must be written in Modula-3, while U-Net/SLE extensions may be written in any language that can be compiled to Java bytecodes. In addition, the use of Java bytecodes in U-Net/SLE allows user extension applets to be transported across the network and run on any network interface with a Java virtual machine, while SPINE's compiled Modula-3 code is architecture-specific. Fourth, SPINE does not define a mechanism by which the network interface can be accessed safely at user level by multiple applications, while U-Net/SLE takes advantage of the existing U-Net user-level network interface architecture. SPINE software was not available at the time of this writing so we cannot make a detailed performance comparison of it to U-Net/SLE. We believe, however, that both SPINE and U-Net/SLE will serve as useful platforms for future research in the areas of user-extensible networks and network interfaces.

6.3 Safe languages

A number of techniques have been suggested for writing "safe" user-level extensions to operating systems. We have already mentioned SPIN, which derives its safety from the type safety of the Modula-3 language and a trusted compiler. VINO [3] uses Software Fault Isolation to protect downloaded kernel extensions from one another. In Proof Carrying Code [13], the operating system publishes a security policy. An extension consists of code and a formal proof that its code does not violate that policy. The operating system verifies the proof; if the proof passes verification, it is accepted as safe and no runtime safety checks are required. Interpreted languages can guarantee safety by providing runtime safety checks in the interpreter; Safe-Tcl [15] is an example of such a language. The Exokernel operating system architecture [9] does not explicitly use a safe language, but allows user-level customization of the operating system by providing a suitably restricted, but very low-level, system interface to applications. ASHs, described earlier, are part of Exokernel. Finally, the Java Virtual Machine enforces safety by using a combination of load-time bytecode verification and runtime safety checks.

Java has received significant attention not only as a language with important safety properties but also as a good candidate for programming embedded devices; indeed, its original target was the embedded processors of set-top television boxes. PersonalJava [19], EmbeddedJava [20], and JavaCard [21] are three Java standards targeted for a range of embedded devices. PersonalJava specifies an API an Application Environment for applications running on networked consumer devices like set-top boxes and PDA's. The PersonalJava API is a subset of the standard Java API plus some new API's. PersonalJava is designed to run on standalone devices with more functionality than network interfaces; for example, it supports the entire Java Virtual Machine and requires support for parts of the Abstract Windowing Toolkit (AWT) API. Neither of these requirements are practical for a virtual machine running on a present-generation network interface.

EmbeddedJava is similar to PersonalJava but is targeted to embedded systems with less processing power and memory, *e.g.* pagers and printers. Like PersonalJava, EmbeddedJava specifies an API and Application Environment. The EmbeddedJava specification has not yet been publicly released.

Unlike PersonalJava and EmbeddedJava, JavaCard specifies not only an API but also a restricted subset of the Java language and Java Virtual Machine. The JavaCard specification describes a Java environment for "smartcards." Like JIVE, JavaCard limits the virtual machine due to memory and processor speed constraints. For example, dynamic class loading, threads, garbage collection, and floating point types are not supported. Unlike JIVE, JavaCard explicitly restricts the supported subset of the Java language; JIVE restricts only the virtual machine model and therefore can run Java bytecodes generated from any language and by any compiler that compiles to that virtual machine subset. Finally, the I/O model for JavaCard differs substantially from that of JIVE: rather than running at the interface between a host and a network, JavaCard *is* the "host," and the card conducts low-bandwidth communication with a Card Acceptance Device (a "smartcard reader").

7 Conclusions and Future Work

We have presented U-Net/SLE, a fast network interface design which permits user extensibility through the downloading of Java applets which run within the network interface itself, and are triggered by transmit and receive events on the network. We believe this design enables a wide range of applications to be built which customize the network interface in order to obtain new communication semantics, more efficiently implement protocols, and reduce host and application overhead by moving elements of protocol processing into the NI.

The performance of our prototype implementation on the Myrinet LanAI processor, while lagging that of the standard U-Net interface, are promising in that the use of interpreted Java bytecodes for packet processing has not resulted in a larger performance penalty (as one might expect). We believe that the use of just-in-time compilation and incorporation of a faster processor onto the network interface will greatly reduce U-Net/SLE overheads and eventually allow the full flexibility of user-safe extensions on the network interface to be realized with minimal overhead.

JIVE, our implementation of the Java Virtual Machine, has demonstrated the difficulty of incorporating the full Java specification into the U-Net/SLE design. Many Java VM features are difficult to implement in an embedded environment, and indeed seem not to be relevant to the needs of small embedded applications. For instance, the use of class inheritance and method overloading are features which can be safely left out of our design. On the other hand, issues such as garbage collection should be addressed. One option would be to circumvent standard garbage collection techniques by defining a persistence model for objects created by a user extension applet (e.g., that arrays created within the `doTx` and `doRx` methods only live until the end of the method invocation).

Another approach which leverages our design is the potential to couple U-Net/SLE with virtual memory extensions in the U-Net model (U-Net/MM [32]). In the U-Net/MM design, applications specify arbitrary virtual memory buffers for transmit and receive, which are translated to physical DMA addresses using a software TLB on the network interface. Providing U-Net/SLE hooks to the TLB lookup and miss-handling process would allow user extension applets to customize the behavior of virtual buffer mapping for network buffers. For example, an application could suggest alternate physical pages to reject when a TLB capacity miss occurs, rather than allowing U-Net/MM to make an arbitrary decision, which is synergistic with with application-level virtual memory systems [8].

In the future we would like to explore the design space of user-programmable network interfaces and I/O controllers in general. Now that our proof-of-concept design has demonstrated the feasibility of user extensibility in the NI, we hope that future implementations will further exploit the benefits of application-customized I/O processing. For example, one could implement the remote memory access operations of the Split-C [2] language directly as a U-Net/SLE extension without requiring the application to implement Active Message handlers for these operations. Understanding the tradeoffs of executing protocol code within the NI as opposed to application level, in general, is an area for future research.

References

[1] C. Bashe, L. Johnson, J. Palmer, and E. Pugh. *IBM's early computers.* Cambridge, MA: MIT Press, 1986.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. "Introduction to Split-C." In *Proceedings of Supercomputing '93.*

[3] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Smith, and D. Tang. "VINO: The 1994 Fall Harvest." Harvard Computer Center for Research in Computing Technology Technical Report TR-34-94, 1994.

[4] M. E. Fiuczynski and B. N. Bershad. "An extensible protocol architecture for application-specific networking." In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[5] M. E. Fiuczynski and B. N. Bershad. "SPINE - A safe programmable and integrated network environment." SOSP 16 Works in Progress, 1997.

[6] S. Goble, D. Barron, M. Bradley, A. Ezzet, and M. Rex. "I2O." Presented at Comdex, Spring 1996. `http://www.i2osig.org/Architecture/I2OArch.Comdex.pdf`

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Reading, MA: Addison-Wesley, 1996.

[8] K. Harty and D. R. Cheriton. "Application-Controlled Physical Memory using External Page-Cache Management." In *Proceedings of ASPLOS*, October, 1992.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, et al. "Application performance and flexibility on exokernel systems." In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.

[10] K. Langendoen, R. Hofman, and H.E. Bal. "Challenging Applications on Fast Networks." Fourth International Symposium on High-Performance Computer Architecture (HPCA-4), Feb. 1-4, 1998 (to appear).

[11] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification.* Reading, MA: Addison-Wesley, 1997.

[12] R. Martin, A. Vahdat, D. Culler, T. Anderson. "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture." International Symposium on Computer Architecture, Denver, CO, June 1997.

[13] G. C. Necula and P. Lee. "Safe Kernel Extensions Without Run-Time Checking." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[14] G. Nelson, editor. *Systems programming with Modula-3.* Englewood Cliffs, NJ: Prentice Hall, 1991.

[15] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. "The Safe-Tcl Security Model." Sun Microsystems Laboratories Technical Report TR-97-60, March 1997.

[16] S. Pakin, M. Lauria, and A. Chein. "High Performance Messaging on Workstations Illinois Fast Messages (FM) for Myrinet." In *Proceedings of Supercomputing '95*, San Diego, California, 1995.

[17] L. Prylli and B. Tourancheau. "Protocol design for high performance networking: a Myrinet experience." Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[18] R. Sedgewick. *Algorithms in C.* Reading, MA: Addison-Wesley, 1990.

[19] Sun Microsystems Inc. "Personal Java 1.0 Specification." `http://java.sun.com/products/personaljava/spec-1-0-0/personalJavaSpec.html`

[20] Sun Microsystems Inc. "Embedded Java." `http://java.sun.com/products/embeddedjava/`

[21] Sun Microsystems Inc. "Java Card 2.0 Language Subset and Virtual Machine Specification." `http://java.sun.com/products/javacard/`

[22] Sun Microsystems Inc. "Java Card 2.0 API." `http://java.sun.com/products/javacard/`

[23] D. L. Tennenhouse and D. J. Wetherall. "Towards an active network architecture." *Computer Communication Review* vol 26 no 2, April 1996.

[24] J. E. Thornton. *Design of a computer: the Control Data 6600.* Glenview, IL: Scott, Foresman and Company, 1970.

[25] *The Unicode Standard: Worldwide Character Encoding,* Version 1.0, Volume 1 and Volume 2. Reading, MA: Addison-Wesley, 1991-1992.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. "U-Net: A User-level Network Interface for Parallel and Distributed Computing." In *Proceedings of the 15th Annual Symposium on Operating System Principles*, December 1995.

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. "Active Messages: A Mechanism for Integrated Communication and Computation." In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[28] D. A. Wallach, D. R. Engler, and M. Frans Kaashoek. "ASHs: Application-Specific Handlers for High-Performance Messaging." In *Proceedings of ACM SIGCOMM '96*, August 1996.

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. "Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, 1993.

[30] M. Welsh, A. Basu, and T. von Eicken. "Low-Latency Communication over Fast Ethernet." In *Proceedings of EUROPAR '96*, August 1996.

[31] M. Welsh, A. Basu, and T. von Eicken. "A Comparison of Fast Ethernet and ATM for Low-Latency Communication." In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, February 1997.

[32] M. Welsh, A. Basu, and T. von Eicken. "Incorporating Memory Management into User-Level Network Interfaces." In *Proceedings of Hot Interconnects V*, August 1997.

[33] T. Wilkinson. "KAFFE: A virtual machine to run Java code." `http://www.kaffe.org/`