U-Net/SLE: A Java-based user-customizable virtual network interface

Matt Welsh [*], David Oppenheimer and David Culler
Computer Science Division, University of California, Berkeley, Berkeley, CA 94720, USA
Tel.:+1 510 643 7566; Fax:+1 510 642 5775;
E-mail: {mdw,davidopp,culler}@cs.berkeley.edu

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, to 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, allows host communication overhead to be reduced and improves the overlap of communication and computation during protocol processing.

1. Introduction

Recent work in high-speed interconnects for distributed and parallel computing systems, 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 [14] and Active Messages [15], virtualize the network interface to provide multiple applications on the same host with direct, protected network access. Other systems, including Fast Messages [10] and BIP [11], 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 [18] to advanced NICs incorporating a programmable co-processor [14].

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 [7] have 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. 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 [5] that in the interest of reducing host overhead, interrupts, and I/O bus transfers, it may be beneficial to perform some protocol processing within the network interface itself, for example on a dedicated network co-processor [4]. 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, eliminating overheads for I/O

[*]Corresponding author.

ISSN 1058-9244 / $8.00 © 1999, IOS Press. All rights reserved

bus transfer and 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 receive addresses.

A number of systems have incorporated these NI-side features in an *ad hoc* manner – 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" 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 Active Messages may use those features without forcing all applications on the same host to go through this interface, while still being able to take advantage of NI-level processing rather than pushing all protocol code to user level.

We have built a prototype of the U-Net/SLE architecture using the Myricom SBus Myrinet interface (which includes a programmable co-processor, the LanAI) and JIVE, a subset of the Java Virtual Machine.

The key contributions of this paper are the definition of a programmatic network interface architecture in which NIC functionality is directly exposed to user applications; the exploitation of this architecture for user customization of network interface behavior; and analysis of Java as a safe language for user extensions executed in the critical path of communication.

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. Section 4 summarizes the performance of our prototype, while Section 5 describes related work. Section 6 concludes and raises issues for future work.

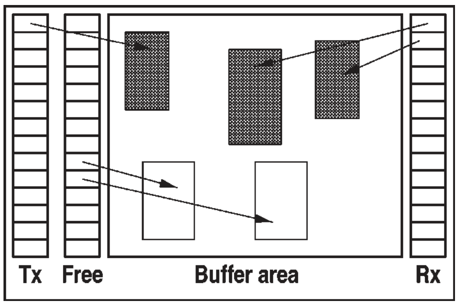

Fig. 1. U-Net endpoint data structure.

2. Design and implementation

U-Net/SLE is based on U-Net [14], 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 Fig. 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, 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

[1]The U-Net/MM architecture [20] extends this model to permit arbitrary virtual-memory buffers to be used.

into the buffer. Once the entire PDU (which may span multiple receive buffers) has arrived, 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. This applet consists of a single class which must implement three methods: a class constructor, `doTx` (invoked when a user pushes an entry onto 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. 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. 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 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. The `doTx` method may then inspect and modify the 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, push descriptors into the user's receive queue, or generate and transmit new network messages. If no applet is associated with this endpoint the data is pushed to the user application as described above.

Native methods are provided for applets to DMA data to and from the user's host buffer area, push a packet to the network, allocate user free buffers, and fill in user receive FIFO entries. The exact nature of these native methods depends on the facilities provided by the NIC; for example, if separate transmit and receive DMA channels are available, they could be managed independently by different native methods.

These native methods enforce protection between user applets and in essence virtualize the network interface resources themselves (such as memory and DMA channels). In effect this technique exposes a *programmatic interface* to the NIC which is a lower-level abstraction than either an endpoint data structure (as in the case of U-Net) or a message-passing API (such as Active Messages). This allows user applications and extension applets to exploit the core subcomponents of the NIC to their immediate advantage, while U-Net/SLE handles virtualization, protection, and resource allocation. Higher-level abstractions (such as message passing, memory channels, and a wide variety of network protocols) can be easily be constructed from the programmatic interface in an application-specific manner; different applications on the same machine can implement different communication abstractions on the same NIC.

2.1. Myrinet prototype 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 of the FORE Systems SBA-200/PCA-200 implementations described in [14,19].

JIVE, our implementation of the Java virtual machine which runs on the LanAI, executes user extension applets in response to transmit request and message receive events. Native methods such as `doHtoLDMA` (DMA data from user to LanAI memory) and `txPush` (push a packet from LanAI memory to the network) are provided which enforce protection between user applets while exposing NIC resources – applets are not allowed to read or write host memory outside of the associated user's U-Net buffer area, for example.

Fig. 2 shows sample U-Net/SLE applet code that implements the standard U-Net mechanism; that is, it

```
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) {  /* Optimization for small msgs */
      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;
  }
}
```

Fig. 2. Sample U-Net/SLE applet source code.

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 4 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 our prototype, Java Implementation for Vertical Extensions (or JIVE).

3.1. JIVE design

JIVE implements a subset of the Java Virtual Machine [6] 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 those found in workstations of the same generation. JIVE classfiles can be generated by any 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. There is no support for `char`, `double`, `float`, or `long`, 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.

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. 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 many useful packet processing applets can be written without them. The current prototype implementation of JIVE does not support garbage collection; while garbage collection is an important issue for future work, we feel that it can be circumvented in this by defining a simple persistence model on created objects (for example, that objects created within the `doTx` or `doRx` methods live only though that method invocation, while those created in the class constructor live for the duration of the class).

The current implementation of JIVE assumes a trusted Java compiler. Bytecode verification should be incorporated into a trusted host daemon that is invoked when a JIVE classfile is loaded into the network interface, thus removing this assumption. 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 [16] or Proof-Carrying Code [8], requires trusting the compiler that generated the code. The Java sandbox, as enforced by the bytecode verifier and runtime checks, protects applets from one another.
- *Speed.* Java bytecode can be interpreted or compiled to native machine code; future work will explore the use of Just-in-Time (JIT) compilation with respect to JIVE.
- *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 because the source and destination are implicit.
- *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 Java virtual machine implementation. A Java classfile could be sent as part of a network packet in an active network [12] 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.

On the other hand, certain Java features are unnecessary for our purposes, such as rich object orientation and threads. We feel that small extension applets running on a network interface can be written without many of the features provided by the Java programming environment.

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 [21], 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.

4. U-Net/SLE prototype performance

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

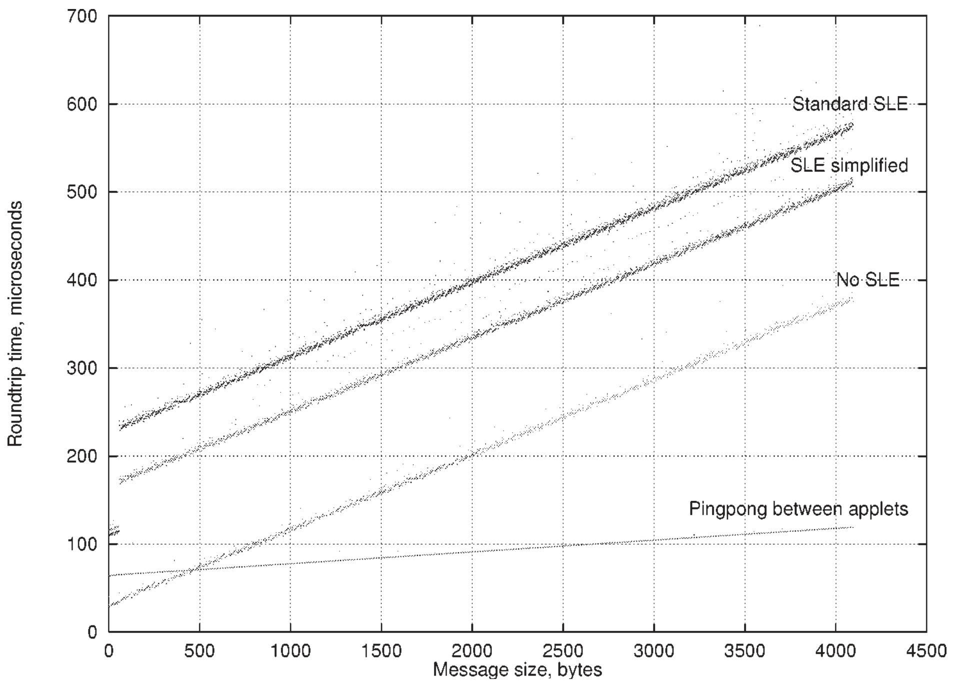

Fig. 3. Round-trip latency vs. message size.

4.1. Latency and bandwidth measurements

Fig. 3 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 Fig. 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.

Fig. 4 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.[2]

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 receive data into its buffer area while performing other computation; no application intervention is necessary to keep the communication 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 inter-

[2]The base U-Net and U-Net/SLE bandwidth can be improved by using both Myrinet DMA engines in parallel; this is not supported by the current LCP.

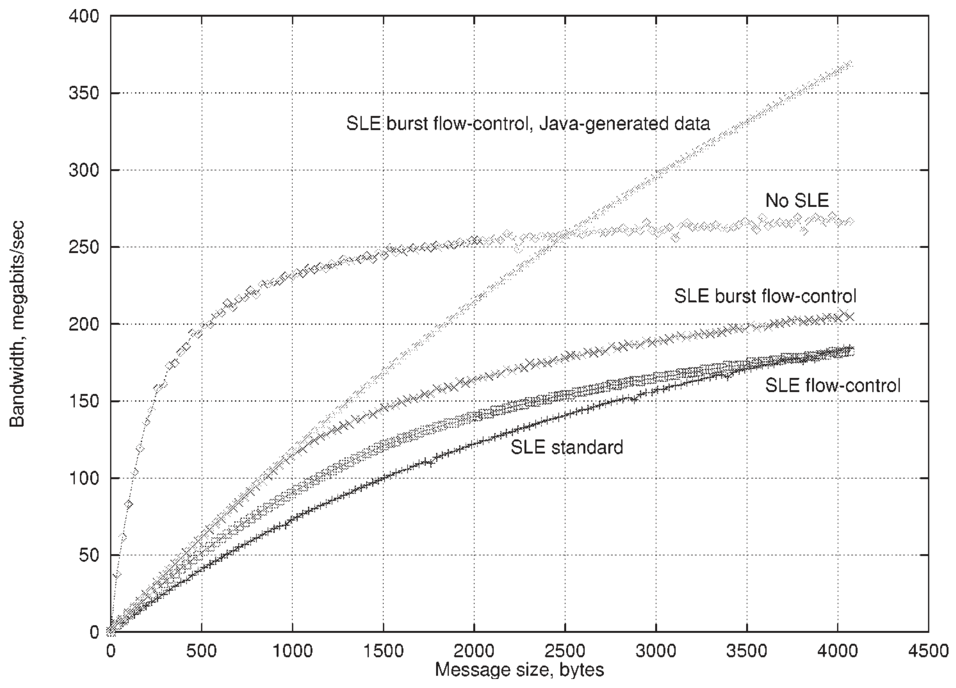

Fig. 4. Bandwidth vs. message size.

face while utilizing SLE extensions for other protocol-processing code.[3]

4.2. *U-Net/SLE overhead breakdown*

Fig. 5 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.

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 (37 MHz) compared to modern workstation 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.

5. Related work

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

5.1. *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

[3]While these micro-benchmarks are unable to directly represent application-specific protocol performance using the SLE extensions, we believe that they characterize the range of performance that can be expected from our prototype. In the final version of this paper we plan to demonstrate higher-level application benchmark results.

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**

Fig. 5. U-Net/SLE transmit/receive operation overhead.

access main memory directly. The Peripheral Control Processors (PPU's) of the CDC 6600 [13] were based on a similar idea. 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. Unlike early programmable I/O controllers, 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.

5.2. *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.

Application Specific Handlers (ASHs) [17] are user-supplied functions that are downloaded into the operating system and are invoked when a message arrives from the network. U-Net/SLE differs from ASHs in several respects. First, ASHs run on the host processor, 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). In addition, 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.

SPIN [3] 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 Modula-3 [9], and the compiler that generates the extensions is trusted to generate non-malicious code.

SPINE [4] 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. 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. Finally, 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. 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. Conclusions and future work

We have presented U-Net/SLE, a fast network interface architecture permitting 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.

U-Net/SLE extends the concept of programmable I/O controllers by exposing a direct programmatic interface to the I/O controller resources; in this case, the DMA channels, memory, and network hardware of the network interface. Pushing the level of virtualization down to the hardware components of the NIC enables a rich set of higher-level abstractions to be constructed which utilize the hardware in an application-specific way.

The performance of our prototype implementation on the Myrinet LanAI processor, while lagging that of the standard U-Net interface, is 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 safe extensions on the network interface to be realized with minimal overhead. We hope to study the use of garbage collection and applet scalability in more detail.

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 execute Active Message handlers for these operations, or implement application-specific protocols (such as video and audio streaming) as a user extension applet. 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*, MIT press, Cambridge, MA, 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*, 1993.

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

[4] M.E. Fiuczynski and B.N. Bershad, Spine – a safe programmable and integrated network environment, in: *SOSP 16 Works in Progress*, 1997.

[5] K. Langendoen, R. Hofman and H.E. Bal, Challenging applications on fast networks, in: *Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

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

[7] R. Martin, A. Vahdat, D. Culler and T. Anderson, Effects of communication latency, overhead, and bandwidth in a cluster architecture, in: *International Symposium on Computer Architecture*, Denver, Colorado, June 1997.

[8] 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.

[9] G. Nelson (ed.), *Systems Programming with Modula-3*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] 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.

[11] 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.

[12] D.L. Tennenhouse and D.J. Wetherall, Towards an active network architecture, *Computer Communication Review* **26**(2) (April 1996).

[13] J.E. Thornton, *Design of a Computer: The Control Data 6600*, Foresman and Company, Glenview, IL, 1970.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Welsh, A. Basu and T. von Eicken, Low-latency communication over fast ethernet, in: *Proceedings of EUROPAR '96*, August 1996.

[19] 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.

[20] M. Welsh, A. Basu and T. von Eicken, Incorporating memory management into user-level network interfaces, in: *Proceedings of Hot Interconnects V*, August 1997.

[21] T. Wilkinson, Kaffe: A virtual machine to run java code, `http://www.kaffe.org/`.