ASHs: Application-Specific Handlers for High-Performance Messaging

Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek, *Member, IEEE*

***Abstract*—Application-specific safe message handlers (ASHs) are designed to provide applications with hardware-level network performance. ASHs are user-written code fragments that safely and efficiently execute in the kernel in response to message arrival. ASHs can direct message transfers (thereby eliminating copies) and send messages (thereby reducing send-response latency). In addition, the ASH system provides support for dynamic integrated layer processing (thereby eliminating duplicate message traversals) and dynamic protocol composition (thereby supporting modularity). ASHs offer this high degree of flexibility while still providing network performance as good as, or (if they exploit application-specific knowledge) even better than, hardwired in-kernel implementations. A combination of user-level microbenchmarks and end-to-end system measurements using TCP demonstrates the benefits of the ASH system.**

***Index Terms*—Computer networks, dynamic code generation, modular computer systems, protocols, operating systems, software protection, user-level networking.**

I. INTRODUCTION

THE COMPLEXITY and ambition of applications scale with increases in processing power and network performance. For example, the last few years have seen a proliferation of distributed shared memory systems [26], [27], [29], real-time video and voice applications [55], parallel applications [10], [40], and tightly coupled distributed systems [2], [48], [52]. Unfortunately, although raw CPU and networking hardware speeds have increased, this increase is not reaching applications: networking software and memory subsystem performance already limit applications and will only do so more in the future [9], [12], [48]. This paper addresses the important problem of delivering hardware-level network performance to applications by introducing application-specific safe message handlers (ASHs), which are user-written handlers that are safely and efficiently executed in the kernel in response to a message arrival. ASHs direct message transfers (thereby eliminating copies), incorporate manipulations such as checksumming and data conversions directly into the message transfer engine (thereby eliminating duplicate message traversals), and send messages (thereby reducing send-response latency). Measurements of a prototype implementation of ASHs demonstrate substantial performance benefits over a high-performance implementation of TCP without ASHs.

ASHs are written by application programmers, downloaded into the kernel, and invoked after a message is demultiplexed (i.e., after it has been determined for which process the message is destined). An important property of ASHs is that they represent bounded, safe computations. ASHs are made safe by controlling their operations and bounding their runtime. Because an ASH is a "tamed" piece of code, it can be directly imported into an operating system without compromising safety. This ability gives applications a simple mechanism with which to incorporate domain-specific knowledge into message-handling routines. ASHs provide three key abilities.

1) Direct, Dynamic Message Vectoring: An ASH can dynamically control where messages are copied in memory, and can therefore eliminate intermediate copies. Systems that do not allow application-directed message transfers often result in messages being copied into at least one intermediate buffer before being placed in their final destination (e.g., an application data structure)

2) Message Initiation: ASHs can send messages. This ability allows an ASH to perform low-latency message replies. The latency of a system determines its performance and scalability; low latency is especially important for tightly coupled distributed systems. For example, one important determinant of parallel program scalability is the latency of communication. In the context of a client/server system, the faster the server can process messages, the less load it has (and, therefore, the more clients it can support) and the faster the response time observed by clients.

3) Control Initiation: ASHs can perform general computation. This ability allows them to perform control operations at message reception, implementing such computational actions as traditional active messages [53] or remote lock acquisition in a distributed shared memory system. Even recently, low-overhead control transfer had been considered to be infeasible to implement [48].

We have also integrated support for *dynamic integrated layer processing* into the ASH system. Current systems often have a number of protocol layers between the application and the network, with each layer often requiring that the entire message be "touched" (e.g., to compute a checksum). Therefore, the negotiation of protocol layers can require multiple

Manuscript received July 31, 1996; revised May 1, 1997; approved by IEEE/ACM TRANSATIONS ON NETWORKING Editor C. Partridge. This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985, by a NSF National Young Investigator Award, by an Intel Graduate Fellowship Award, and by Digital Equipment Corporation. This paper was presented at ACM SIGCOMM'96.

D. A. Wallach is with the Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA 94301 USA (e-mail: kerr@pa.dec.com).

D. R. Engler and M. F. Kaashoek are with the Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139 USA (e-mails: engler@lcs.mit.edu; kaashoek@lcs.mit.edu).

Publisher Item Identifier S 1063-6692(97)06044-5.

costly memory traversals, stressing a weak link in high-performance networking: the memory subsystems of the end-point nodes. As argued by Clark and Tennenhouse [9], an *integrated* approach, where these application-specific operations are combined into a single memory traversal, can greatly improve the latency and throughput of a system.

The ASH system integrates data manipulations such as checksumming or conversions into the data transfer engine itself, automatically and dynamically performing integrated layering processing (ILP). Even though ASHs improve flexibility by using layers integrated at runtime, dynamic ILP is as efficient as statically written hard-wired ILP implementations.

The ASH system has been implemented in Aegis, an exokernel operating system [20] for MIPS-based DECstations. We examine two network devices: a 10-Mb/s Ethernet and a 155 Mb/s AN2 (Digital's ATM network). On top of the raw network interface we have implemented several network protocols (ARP/RARP, IP, UDP, TCP, HTTP, and NFS) as user-level libraries, which are then linked to applications. We demonstrate that these user libraries perform well and are competitive with the best systems reported in the literature. We use a combination of user-level microbenchmarks and end-to-end system measurements to demonstrate the benefits of the ASH system.

The remainder of the paper is structured as follows. Section II discusses design issues and Section III, implementation issues for ASHs. Section IV describes the experimental environment we use to evaluate the benefits of ASHs. Section V reports on how ASHs can be used and illustrates their benefits. Section VI relates ASHs to other work. In Section VII, we draw our conclusions.

II. Types of ASHs

Once a message is demultiplexed to a particular application, the message must be delivered to it. There are a variety of actions that can be required in a networking system: message vectoring (e.g., copying a message into its intended slot in a matrix), message manipulations (e.g., checksums), message initiation (e.g., message reply), and control initiation (e.g., computation). An application-specific safe message handler (ASH) can perform all of these operations.

ASHs are user-written routines that are downloaded into the kernel to efficiently handle messages. From the kernel's point of view, an ASH is simply code, invoked upon message arrival, that either consumes the message it is given or returns it to the kernel to be handled normally. From a programmer's perspective, an ASH is a routine written in a high-level language and potentially augmented with pipes for dynamic ILP, or it is a series of routines representing protocol layers which will be composed together.

Operationally, ASH construction and integration has three steps. First, client routines are written using a combination of specialized library functions and any high-level language that adheres to C-style calling conventions and runtime requirements. These routines, in the form of machine code, are then handed to the ASH system. The ASH system post-processes this object code, ensuring that the user handler is safe through a combination of static and runtime checks, and downloads it into the operating system, handing back an identifier to the user for later reference. The user can then use the identifier to associate the ASH with a user-specified demultiplexor. When the demultiplexor accepts a packet for an application, the ASH will be invoked. The ASH can then control where to copy the message, integrate data manipulations into this copy, and/or send messages.

Many of the benefits of ASHs can be obtained with a relatively small amount of support software. The simplest form of ASHs is *static* ASHs, e.g., ASHs with no dynamic code generation support. Without this support, they cannot take advantage of dynamic ILP or dynamic protocol composition. With just the addition of *pipes*, as explained below, they can use dynamic ILP. The further addition of protocol composition greatly eases the task of writing ASHs, allowing protocol fragments to be dynamically and modularly built and composed, at the cost of requiring a fair amount of support software. This paper explores the use of ASHs in the first two cases: for applications which do not require much data manipulation, tiny, extremely fast hand-written static ASHs are most appropriate; ASHs using dynamic ILP, on the other hand, are more useful for latency-critical applications which perform a lot of data manipulation. We describe each of these types of ASHs in turn.

A. Static ASHs

All types of ASHs are generally written in a stylized form consisting of three parts. The initial part consists of protocol and application code that examines the incoming message to determine if the ASH can be run and where the data carried by the incoming message should be placed. The second part is the data manipulation part; the data is manipulated as it is copied from the message buffer (or left in place, if desired). Static ASHs are responsible for hand-orchestrating any data manipulations they require. If there are several actions to be taken here, such as a checksum and a copy, the other forms of ASHs can use integrated layer processing at this point. The third and final part again consists of protocol and application code, of two types: *abort* and *commit*. Which of these is run depends on the initial code and possibly the result of the data manipulation step.

If the first two parts complete successfully, the commit code is called. The commit code performs any operations indicated by the incoming message, including, if appropriate, initiating a message or performing computation. If the ASH detects that something went wrong, on the other hand (for example, a needed lock could not be acquired), it calls its abort code to fix up any state that has been modified. We refer to this as a *voluntary* abort: the ASH writer is responsible both for detecting the problem and for fixing up any changes the ASH has made at that point.

B. ASHs with Dynamic ILP

Simple static ASHs can be extended to use the dynamic ILP support provided by the ASH system. In addition to simple data copying, many systems perform multiple tra-

```
pl = pipel(2);                          // Initialize a pipelist for two pipes (checksum and byteswap)
checksum_pipe_id = mk_cksum_pipe(pl, &pipe_cksum);      // Create checksum pipe
byteswap_pipe_id = mk_byteswap_pipe(pl);                // Create byteswap pipe
ilp = compile_pl(pl, PIPE_WRITE);   // Compile the two pipes, return a handle to the integrated function
```

Fig. 1. Compose and compile checksum and byteswap pipes.

```
// Specify a pipe to compute the Internet checksum and return its identifier.
// This specification is subsequently converted into safe machine code.
// This code assumes that messages are always a multiple of four bytes long.
int mk_cksum_pipe(struct pipel *pl, reg_type *cksum_reg) {
   reg_type reg;
   int pipe_id;

   // This 32-bit checksum is commutative and does not alter its input.
   pipe_lambda(pl, &pipe_id, P_GAUGE32, P_COMMUTATIVE | P_NO_MOD);
      reg = p_getreg(pl, pipe_id, P_VAR); // Allocate an accumulate register (preserved across pipe applications)
      p_input32(p_inputr);              // Get 32 bits of input from the pipe
      p_cksum32(reg, p_inputr);         // Add input value to checksum accumulator
      p_output32(p_inputr);             // Pass 32 bits of output to next pipe
   pipe_end();
   *cksum_reg = reg;
   return pipe_id;
}
```

Fig. 2. Simple checksum pipe example.

versals of message data as every layer of the networking software performs its operations (e.g., checksumming, encryption, conversion). At an operational level, these multiple data manipulations are as bad as multiple copies. To remove this overhead, Clark and Tennenhouse [9] propose *integrated layer processing* (ILP), where the manipulations of each layer are compressed into a single operation.

To the best of our knowledge, all systems based on ILP are static, in that all integration must be hard coded into the networking system. This organization has a direct impact on efficiency: since untrusted software cannot augment these operations, any integration that was not anticipated by the network architects is penalized. Furthermore, many systems compose protocols at runtime [5], [24], [25], [50], [51], making static ILP infeasible. There are additional disadvantages to static ILP: static code size is quite large, since it grows with the number of *possible* layers instead of actually used layers, and augmenting the system with new protocols is a heavyweight operation that requires, at least, that the system be recompiled to incorporate new operations. Therefore, we designed the ASH system to support dynamic ILP. Note that previous systems have also found dynamic code generation useful in eliminating the cost of modularity in other areas [33].

ILP can be dynamically provided through the use of *pipes*, which were first proposed by Abbott and Peterson [1] for use in static composition. A pipe is a computation written to act on streaming data, taking several bytes of data as input and producing several bytes of output while performing only a tiny computation (such as a byteswap, or an accumulation for a checksum). The ASH pipe compiler dynamically integrates several pipes into a tightly integrated message transfer engine which is encoded in a specialized data copying loop.

To allow modular coupling, each pipe has an input and output gauge associated with it (e.g., 8 b, 32 b, etc.). This allows pipes to be coupled in a distributed fashion; the ASH system performs conversions between the required sizes. For example, a checksum function may take in and generate 16-b words, while an encryption pipe may require 32-b words. To allow a 16-b checksum pipe's output to be streamed through a 32-b encryption pipe, it is aggregated into a single register.

Fig. 1 presents an example composition of a checksum pipe (described below) with a pipe to swap bytes from big to little endian. There are two important points in this figure. First, the composition is completely dynamic: any pipe can be composed with any other at runtime. Second, it is modular: the ASH system converts between gauge sizes and prevents name conflicts by binding the context inside the pipe itself.

The pipes for ASHs are written in VCODE [18], which is a set of C macros that provide a low-level extension language for dynamic code generation. VCODE is designed to be simple to implement and efficient both in terms of the cost of code generation and in terms of the computational performance of its generated code.

The VCODE interface is that of an extended RISC machine: instructions are low-level register-to-register operations. A sample pipe to compute the Internet checksum [6] is provided in Fig. 2. Each pipe is allocated in the context of a pipe list (`pl` in the figure) and given a pipe identifier that is used to name it. Additionally, pipes are associated with a number of attributes controlling the input and output size (a pipe's "gauge"), whether the pipe is allowed to transform its input, and whether the pipe is commutative (i.e., whether it can perform operations on message data out of order). These attributes govern how a given pipe is composed with other pipes (e.g., whether it can be reordered, and the expected input and output sizes) and how it can be used.

Since pipe operations are written in terms of portable assembly language instructions, pipes are charged with allocating those registers they need and choosing the appropriate register class. The two available register classes are *temporary* and

persistent. Temporary registers are scratch registers that are not saved across pipe invocations. Persistent registers are saved across pipe invocations; they are used, for example, as accumulators during checksum computations (`cksum_reg` in Fig. 2). The values of persistent registers can be imported and exported from the main protocol code. *Export* is used to initialize a register before use, and *import* to obtain a register's value (e.g., to determine if a checksum succeeded). The special register `p_inputr` is reserved to indicate the pipe's input.

Since current compilers do not optimize networking idioms well, we have extended the VCODE/system to include common networking operations. Current compilers lack clear idiomatic ways of expressing common networking operations such as checksumming, byteswapping, memory copies, and unaligned memory accesses from within even a low-level language such as C. To remedy this situation, the VCODE extensions offer primitives for expressing common networking operations. If desired, clients of the ASH system can extend VCODE with further primitives with little performance impact.

Fig. 2 exploits the extension added to VCODE for computing the Internet checksum. On machines such as the SPARC and the Intel x86, this pipe is compiled by VCODE to use the provided add-with-carry instructions to efficiently compute the checksum a word at a time. In the given example, the pipe consumes a 32-b word of data (using the `p_input32` instruction), adds its value to the running checksum total along with any overflow (using the `p_cksum32` instruction), and then outputs the unchanged input. The ASH that calls this function is responsible for setting up the initial state of the accumulator register, then later reading it in, and folding it to 16 b.

C. Dynamic Protocol Composition

In addition to dynamic ILP, ASH programmers can use the dynamic protocol composition extensions provided by the ASH system [21]. Whereas dynamic ILP provides modularity in terms of pipes (only one checksum routine has to be written, and can be composed with any other routine), dynamic protocol composition provides modularity in terms of entire protocols (only one IP routine has to be written, and can be composed with UDP or TCP). The initial results [56] are promising, but due to space constraints a complete description is beyond the scope of this paper.

III. Implementation

All varieties of ASHs require a certain amount of system support, both to run correctly and to prevent them from damaging the operating system or other applications. This section describes the support: the requirements we place on the operating system, our strategies for executing ASHs safely, and the additional support for dynamic ILP.

A. The Operating System Model

The most important task required of the operating system is that it allows an ASH to execute in the addressing context of its associated application. Without this ability, ASHs would be difficult to write, since they could not directly use application data structures.

In the MIPS architecture on which we have developed the ASH system, supporting address translation is fairly simple: before initiating an ASH, the context identifier and pointer to the page table of its associated application must be installed. As described below, when an ASH references a nonresident page, or an illegal memory address, it is aborted.

Note that address translation need not be dynamic. To eliminate translation faults, the physical address a virtual address maps to can be pre-bound into the ASH when it is imported into the kernel. This paper does not explore this methodology.

Secondary functions that the operating system should provide (but are orthogonal to our discussion) include memory allocation, page-protection modification, and creation of virtual memory mappings. To increase the likelihood that memory will be resident when messages arrive, there should be a mechanism by which applications can provide hints to the operating system as to which pages should remain in main memory. Applications may also want to be able to influence the scheduling policies.

We do not assume that the operating system can field ASH-induced exceptions (such as arithmetic overflow), that ASHs necessarily have access to floating point hardware, or that hardware timing mechanisms are available. As discussed in Section III-B, we have designed safety provisions to remove the necessity for these functions.

In our current system, we require that the application pin all pages that the ASH may reference. A reference to an absent page causes the ASH to be terminated. We plan to provide the ability to suspend ASH handlers and later restart them (still in the kernel) at some point in the future. This addition will require that the OS be able to save all of an ASH's state, including its live registers and the message that it was processing.

B. Safe Execution

We use various techniques to detect malicious or buggy ASH operations (such as divide by zero or excessive execution), and either prevent them or terminate the ASH with an *involuntary abort* if they occur. Because such operations indicate incorrect code, our concern is only to prevent damage to the operating system or to other processes, and thus the application that installed the ASH may no longer operate correctly after an involuntary abort (i.e., if the ASH had made some modifications to application data structures before being terminated).

For safety, the ASH system must guard against exceptions, wild memory references and jumps, and excessive execution time by ASHs. There are various ways to guarantee safety, depending on the hardware platform being used. For example, the implementation of static ASHs for the Intel x86 uses hardware support for segmentation and privilege rings to guard ASHs; in this implementation almost no software checks are needed.[1] The MIPS implementation, in contrast, must use

[1] David Mazières from MIT designed and implemented the x86 version.

software techniques. We describe these software techniques here in detail.

1) Preventing Exceptions: Exceptions are prevented either through runtime or download-time checks. Runtime checks are used to prevent divide-by-zero errors; unaligned exceptions are prevented by forcing pointers to be aligned to the requirements of the base machine[2]. Arithmetic overflow exceptions can be prevented by converting all signed arithmetic instructions to unsigned ones (which do not raise overflow exceptions) or code using them may be disallowed (as is currently done, because the C compiler that we use never generates any signed arithmetic instructions). At download time, we prevent the usage of floating-point instructions. Many of these checks could be removed in a more sophisticated implementation that had operating system support for handler exceptions. With such support, we could optimistically assume exceptions would not happen: if any did occur, the kernel would then catch them and abort the ASH.

Address translation exceptions are handled by the operating system. In the case of a TLB refill, the operating system replaces the required mapping and resumes execution. In the case of accesses to nonresident pages or illegal addresses, the ASH is aborted.

2) Controlling Memory References: Addressing protection is implemented through a combination of hardware and software techniques. Wild writes to user-level addresses are prevented using the memory-mapping hardware. As discussed above, when an ASH is initiated, its context identifier and page table pointer are installed; additionally, it is run on a user-level stack.

On the MIPS architecture, code executing in kernel mode can read and write physical memory directly. To prevent this, we force all loads and stores to have user-level addresses, using the code modification (*sandboxing*) techniques of Wahbe *et al.* [54].

Making sandboxed data copies efficient requires complex analysis of the user-supplied code. The ASH system therefore uses semantics to obtain efficiency by providing the capability of accessing message data through specialized trusted function calls, implemented in the kernel. These calls allow access checks to be aggregated at initiation time. Experiments show that these checks add little to the base cost of data transfer operations. Also, message data may be efficiently moved using the dynamic ILP support.

Wild jumps are also prevented by code inspection. All indirect jumps are checked at runtime. If they are to somewhere within the current ASH they proceed unchanged; if they are to code named by the pre-sandboxed address then they are translated and allowed to proceed; if they are to operating system calls explicitly allowed by the system (such as the network send system call) they are called directly; if they are to anywhere else the ASH is aborted immediately.

3) Bounding Execution Time: Because we want to allow 4-kbyte messages to be copied, decrypted, and checksummed, the instruction budget of the ASHs we describe in this paper is rather large (tens of thousands of instructions). For ASHs which contain no loops (or loops bounded only by the message size), we can simply overestimate the effects of straight-line code to create overly pessimistic, but simple to implement estimations of execution time.

For ASHs that contain loops, software checks at all backward jump locations need to be inserted. On machines with appropriate hardware, cycle counters can aid in generating efficient execution time accounting code. Systems with timers can be exploited to remove all software checks. Our prototype uses the third approach, aborting any ASH that attempts to use two clock ticks worth of time or more. Setting up and clearing these timers takes approximately one microsecond each on our system. ASHs are never aborted during system calls (the abort is delayed until the ASH exits the system call).

C. Dynamic ILP Interface

As discussed in Section II-B, an application specifies the data manipulation steps it wants integrated in VCODE. It presents this list to the dynamic ILP (DILP) system, which compiles the list down into machine code representing the requested list of data manipulations, and returns an identifier (`ilp` in Fig. 1). When an ASH then wishes to perform those steps, it presents this identifier, a pointer to source data, length, and possibly a pointer to destination data to the DILP engine, which calls the generated specialized data copying and manipulation loop. Different loops may be generated for different network interfaces; for example, our Ethernet DMA engine stripes an N-byte contiguous packet into a $2N$-byte buffer, alternating 16 bytes of data and 16 bytes of padding, whereas the AN2 DMA engine copies the data contiguously.

Some network interfaces provide the capability of having part of the message read by the processor, and then allowing the rest of the message to be written directly from the NI to where the message directs. Although not part of our current implementation, ASHs can take advantage of this type of interface as well, in an especially clean manner if done through the DILP interface (from the point of view of the application, nothing except for performance should change). Only the back end of the DILP engine should have to change.

IV. Experimental Environment

This section reports on the base performance of our system (i.e., without ASHs). The next section reports on the benefits of using ASHs. Like other systems [15], [16], [31], [49], [52], all the protocols are implemented in user space. The main point to take from the results in this section is that our implementation performs well and is competitive with the best systems reported in the literature. We will discuss in turn the testbed, the raw performance of the network system, and the performance of our user-level implementations of UDP and TCP.

A. Testbed

We have implemented a system for ASHs in an exokernel operating system [20]. Although our implementation is for an exokernel, ASHs are largely independent of the specific operating system and operating system architecture. They

[2] This particular check is not yet implemented, but is straightforward to add.

should apply equally well to monolithic and microkernel systems. Similarly, they apply equally well to in-kernel (e.g., TCP/Vegas), server (e.g., Mach network server), or user-level (e.g., U-Net) implementations of networking.

Aegis, an exokernel for MIPS-based DECstations, provides protected access to two network devices: a 10-Mb/s Ethernet and a 155-Mb/s AN2 (Digital's ATM network). The Ethernet device is securely exported by a packet filter engine [35]. The Aegis implementation of the packet filter engine, DPF [19], uses dynamic code generation. DPF exploits dynamic code generation in two ways: by using it to eliminate interpretation overhead by compiling packet filters to executable code when they are installed into the kernel, and by using filter constants to aggressively optimize this executable code. DPF is an order of magnitude faster than the highest performance packet filter engines (PATHFINDER [3] and MPF [57]) in the literature.

Similarly to other systems [15], [40], [52], the AN2 device is securely exported by using the ATM connection identifier to demultiplex packets. Before communicating, processes bind to a virtual circuit identifier, providing a section of their memory for messages to be DMA'ed to. The kernel and user share a virtualized notification ring per virtual circuit; by examining this ring an application can determine that a message arrived and where the message was placed. The application is allowed to use those message buffers directly, as long as it eventually returns or replaces them. The buffers are guaranteed never to be swapped out in our current implementation. Other organizations are possible; Smith *et al.* discusses several alternatives in [43].

The measurements in this paper are taken on a pair of 40-MHz DECstation 5000/240s, which are rated at 42.9 MIPS and 27.3 SPECint92. The 240 has separate direct-mapped write-through 64-kbyte caches for instructions and data. The I/O devices are accessed over a 25-MHz TURBOchannel bus. The two DECstation 240s are connected with an AN2 switch.

B. Methodology

While collecting the numbers reported in this paper, we had a fair number of problems with cache conflicts (similar to problems reported by others [36]), because the DECstations have direct-mapped caches. In order to minimize the effect these conflicts had on our experiments, we automatically linked the kernel object files in many different orders and picked a best-case timing to report, for every application. We feel that this methodology provided a fair comparison between the different experiments.

From a given run to run, the numbers reported in this paper stayed fairly constant. We used both the system elapsed time as measured by clock interrupt and a cycle counter located on the AN2 board to measure application run times; as long as a given application ran long enough, these two numbers matched. Except as specially noted all of the exokernel numbers were taken as follows. We ran multiple iterations of each experiment (so the application ran long enough to measure) and divided by the number of iterations to calculate the run time. For each experiment, we took ten such data points and calculated 95% confidence intervals from them. In all cases, the size of the confidence intervals was less than 3 μs, and they are not reported here.

TABLE I
RAW LATENCY (IN MICROSECONDS PER ROUND TRIP) FOR USER-LEVEL AND IN-KERNAL APPLICATIONS ON AN2 AND ETHERNET

Network	Latency
in-kernel AN2	112
user-level AN2	182
Ethernet	309

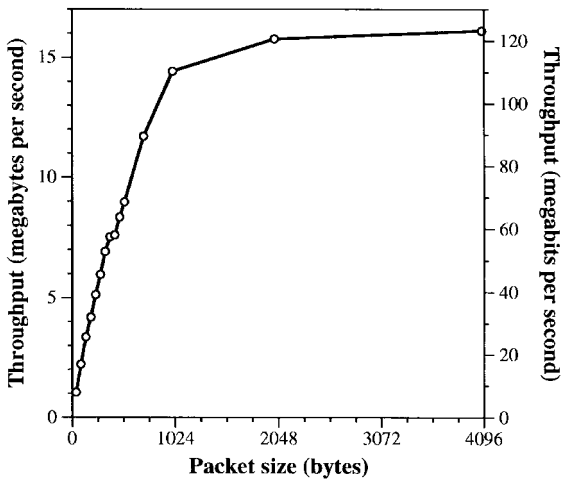

Fig. 3. Throughput for a user-level application on the AN2.

C. Raw Performance of Base System

The raw performance of our base system, i.e., without the use of ASHs, is competitive with other highly optimized systems employing similar hardware. Table I shows the roundtrip latency achieved using the Ethernet and AN2 interfaces to send and receive from user space a 4-byte message between two DECstation 5000/240s.

For the AN2 interface, the table also compares the user-level version to the best in-kernel version we were able to write. Since the hardware overhead for a round trip is approximately 96 μs [40], the kernel software is adding only 16 μs of overhead. The user-level number, which includes the time to schedule the application, cross the kernel-user boundary multiple times, and use the full system call interface, adds another 70 μs, which brings the total software overhead to 86 μs, or about 3440 cycles. For this measurement, the user-level application is sitting in a tight loop polling for a message; the other processes on the system are basically idle.

Fig. 3 is a graph of the bandwidth obtainable in our system by sending a large train of packets of different sizes from user level. The maximum achievable per-link bandwidth is about 16.8 Mbytes/s (134 Mb/s) [40]. At a 4-kbyte packet size, we reach 16.11 Mbytes/s.

These raw numbers are competitive with other high-performance implementations that also export the network to user space. Scales *et al.* [40] measure about twice as much software overhead (7600 cycles or 34 μs) for a null packet send using their `pvm_send` and `pvm_receive` interface using the same ATM board, with a substantially faster machine

TABLE II
LATENCY AND THROUGHPUT FOR UDP AND TCP OVER AN2 AND ETHERNET. THE LATENCY IS MEASURED IN MICROSECONDS, AND THE THROUGHPUT IN MEGABYTES PER SECOND

	UDP		TCP	
Implementation	**Latency**	**Throughput**	**Latency**	**Throughput**
AN2; in place, no checksum	221	11.69	333	5.76
AN2; in place, with checksum	244	7.86	383	4.42
AN2; no checksum	225	8.57	333	5.02
AN2; with checksum	244	6.45	384	4.11
Ethernet; with checksum	309	1.02	443	1.03

(a 225-MHz DEC 3000 Model 700 AlphaStation rated at 157 SPECint92). Our absolute numbers are higher than U-Net (182 μs versus 66 μs), since our experiments are taken on slower machines (40-MHz versus 66-MHz), the AN2 hardware latency is higher than the Fore latency (96 μs versus 42 μs), and we have not attempted to rewrite the AN2 firmware to achieve low latency, as was done for U-Net [52]. Direct comparisons with other high-performance systems such as Osiris [15] and Afterburner [16], [17] are difficult since they run on different networks and have special purpose network cards, but our implementation appears to be competitive.

D. User-Level Internet Protocols

On top of the raw interface we have implemented several network protocols (ARP/RARP, IP, UDP, TCP, HTTP, and NFS) as user-level libraries, which are then linked to applications. The general structure is similar to other implementations of user-level protocols [16], [52]. The UDP implementation is a straightforward implementation of the UDP protocol as specified in RFC768. Similarly, the TCP implementation is a library-based implementation of RFC793. We stress that the TCP implementation is not fully TCP compliant (it lacks support for fluent internetworking such as fast retransmit, fast recovery, and good buffering strategies). Nevertheless, both the UDP and TCP implementations communicate correctly and efficiently with other UDP and TCP implementations in other operating systems.

Table II shows the latency and throughput for four different implementations of UDP and TCP over the AN2 and the Ethernet. On the AN2, the TCP implementation uses the virtual circuit identifier and the ports in the protocol header to demultiplex the message to the desired protocol control block; the UDP implementation currently uses only the virtual circuit index. As observed by many others, user-level protocols provide opportunities for optimization not necessarily available nor convenient for traditional in-kernel protocols. These tables demonstrate the benefits achievable through the use of these optimizations. The AN2 *in place, no checksum* measurements demonstrate the best performance we have achieved for UDP and TCP implemented as user-level protocols. In this case, there are no additional copies from the network interface to application data structures and the implementation relies on the CRC computed by the AN2 board for checksumming. To simulate the lack of additional copies, the code throws away the application data in the *in place* versions (this zero copy can be achieved; with our user-level AN2 interface (similar to others, e.g., Smith *et al.* [43]) an application can be informed where its data has landed, and may use the data directly out of that buffer as long as it replaces the buffer with some other one). For the non-*in place* versions of our measurements, the application and the protocol library are separated by a traditional read and write interface, resulting in an additional copy between the network and application data structures. We also present measurements with end-to-end checksumming. In the *with checksum* measurement, the protocol library copies the data from the network to the application data structures and also computes the Internet checksum. This last configuration is closest to what one might expect from a hard-coded in-kernel implementation.

The left most columns of Table II show the latency and throughput for different implementations of UDP over AN2 and Ethernet. Latency is measured by ping-ponging 4 bytes. Throughput is measured by sending a train of six maximum-segment-size packets (1500 bytes for Ethernet and 3072 bytes for AN2) and waiting for a small acknowledgment. Using larger train sizes increases the throughput.

On the Ethernet, both UDP latency and throughput are (modulo processor speed differences) about the same as the fastest implementation reported in the literature [47]. On the AN2 interface, UDP latencies are about 43 μs higher than the raw user-level latencies. This difference is because the UDP library allocates send buffers, and initializes IP and UDP fields. Our implementation seems to have lower overhead than U-Net; the U-Net implementation adds 73 μs on a 66-MHz processor while our implementation adds 62 μs on a 40-MHz processor (even though, unlike their numbers, our checksum and memory copy are not integrated for this measurement) [52]. In contrast, UDP running over Ultrix on our platform requires about 1500 μs per round trip. The bandwidth is mostly a function of the train size used in the experiment. With a large enough train the UDP experiment achieves nearly the full network bandwidth.

The right most columns of Table II show the latency and throughput for different implementations of TCP over AN2 and Ethernet. Latency is measured by ping-ponging 4 bytes across a TCP connection. Throughput is measured by writing 10 Mbytes in 8-kbyte chunks over the TCP connection. For the AN2 the maximum segment size is 3072 bytes and for the Ethernet the maximum segment size is 1500 bytes. For both networks the window size was fixed at 8 kbytes to ensure experiment repeatability. Larger window size increases the throughput. Except during connection set up and tear down, all segments were processed by the TCP header-prediction code.

The difference between UDP and TCP latency is mostly due to the fact that the write call (i.e., sending) is synchronous

(i.e., write waits for an acknowledgment before returning); as a result the data that is piggybacked on the acknowledgment has to be buffered until the client calls read (which leads to an additional copy in our current implementation). In addition, the overhead of returning out of the write call and starting the read call cannot be hidden. Finally, there is some amount of nonoptimized protocol processing (checking the validity of the segment received and running header-prediction code). These sources of overhead, together accounting for about 140 μs, seem also to account for most of the difference in latency with U-Net, which adds a total of 20 μs (on a 66-MHz machine) over their UDP implementation.

In summary, the base performance of our system for UDP and TCP is about the same or is better than most high-performance user-level and in-kernel implementations [15], [16], [22], [31], [49], as long as the applications are scheduled when the messages arrive.

V. Using ASHs

In this section, we examine the specific benefits that application-specific safe handlers enable: high throughput, low-latency data transfer, and low-latency control transfer. We show that ASHs can achieve better performance than polling, even when there is a single active process on the system, and furthermore that when there are multiple processes on the system, ASHs provide higher performance than simple user-level networking.

We implemented fast asynchronous upcalls to compare ASHs with. Upcalls involve application code (a handler) being run at user level in response to a message. Because this code is not being downloaded into the kernel, it does not need to be made safe. Although an upcall requires a switch to user space to run the handler, a full process switch is unnecessary; this allows upcall invocation to be fast (though not as fast as ASH invocation, as we will show).

We use a combination of user-level microbenchmarks and end-to-end microbenchmarks. The user-level measurements gauge the individual effects of, for example, avoiding copies, while the end-to-end measurements give insight into the end-to-end performance effects. The user-level microbenchmarks measure throughput in megabytes per second for operations performed on 4096 bytes of data. We assume that the message and its application-space destination are not cached when the message arrives, and so perform cache flushes at every iteration. The network send and receive buffers are modeled as simple buffers in memory.

The end-to-end measurements are taken on the system described in the previous section. Because TCP is important, well documented, and widely used, we try to illustrate the benefits of ASHs using TCP. In order to separate out the cost of sandboxing, we report experimental results both with and without the cost of sandboxing overhead (although in both cases the ASH is prevented from running too long by a timer).

Similarly to how applications running on Aegis are given the entire message to process (after a demultiplexing step based on virtual circuit identifier), so too are ASHs. No higher level demultiplexing is done; no higher-level protocol (e.g., IP) is forced upon ASHs. Similarly, for the Ethernet implementation reported in [20], demultiplexing of a message to an ASH was done through DPF (see Section IV); again, no more functionality is required in the kernel than is needed to demultiplex the messages to the correct process in the first place. Note that ASHs are invoked directly from the AN2 device driver, just after it performs a software cache flush of the message location, to ensure consistency after the DMA.

It should be noted that the results of our experiments underestimate the benefits of running ASHs in any other kernel because kernel crossings in Aegis have been highly optimized: Aegis' kernel crossing times are five times better than the best reported numbers in the literature and are an order of magnitude better than a run-of-the-mill UNIX system like Ultrix [20]. For example, on the DECstation 5000/240 the advantage of running an ASH in the Aegis exokernel versus running an upcall in user space is approximately 35 μs; under Ultrix4.2 this difference would be more like 95 μs (the approximate cost of an exception plus the system call back into the kernel) [20].

A. High Throughput

High data transfer rates are required by bulk data transfer operations. Unfortunately, while network throughput and CPU performance have improved significantly in the last decade, workstation memory subsystems have not. As a result, the crucial bottleneck in bulk data transfer occurs during the movement of data from the network buffer to its final destination in application space [9], [12]. To address this bottleneck, applications must be able to direct message placement, and to exploit ILP during copying. We examine each below.

1) Avoiding Message Copies: Message copies can cripple networking performance [1], [9], [48]. However, most network systems do not allow applications to direct data transfer. This results in needless data copies as incoming messages are copied from network buffers to intermediate buffers (e.g., BSD's mbufs [28]) and then copied to their eventual destination. To solve this problem, we allow a handler to control where messages are placed in memory, eliminating all intermediate copies. Our general computational model provides two additional benefits. First, these data transfers do not have to be "dumb" data copies: handlers can employ a rich "scatter-gather" style, and use dynamic, runtime information to determine where messages should be placed, rather than having to pre-bind message placement. Second, in the context of a highly active gigabit per second network, tardy data transfer can consume significant portions of memory for buffering: the quick invocation of handlers allows the kernel buffering constraints to be much less.

Copying messages multiple times dramatically reduces the maximum throughput. We can see this by measuring the time to: 1) copy data a single time; 2) copy data two times, where the data is in the cache for the second copy; and 3) copy data twice, where the data is not in the cache for the second copy. Table III demonstrates that a second copy degrades throughput by a factor of 1.4 for cached data, and by a factor of two for uncached, as expected. We also observe this effect in the

TABLE III
THROUGHPUT FOR COPIES OF 4096 BYTES OF DATA: SINGLE COPY, TWO CONSECUTIVE COPIES (DATA IN CACHE), TWO CONSECUTIVE COPIES WITH INTERVENING CACHE FLUSH. THROUGHPUT IS MEASURED IN MEGABYTES PER SECOND

single copy	double copy	double copy (uncached)
20	14	11

Aegis UDP and TCP implementations: the throughput for the *no checksum* version of UDP increases by a factor of 1.1–1.4 when the copy from the network buffers into the application's data structures is eliminated, as was shown in Table II.

Our system's data transfer mechanism enables applications to exploit the capabilities of the network interface in avoiding data transfer. For interfaces such as the Ethernet, the network buffers available to the device to receive into are limited, and therefore a message must not stay in them very long. In this case, at least one copy is always necessary. Through the use of a handler, the application can ensure that the copy is to its own data structures, and that no further copies are needed. The AN2 network interface card, on the other hand, can DMA messages into any location in physical memory. An application which does not need to move message data into its own data structures, but which can instead use it wherever it has landed, can take advantage of this feature and avoid all copies. Applications which require that the data be copied, on the other hand, can use handlers to do so; furthermore, through the use of dynamic ILP, they can ensure that the copy is integrated with whatever other data manipulation may be required.

2) Integrated Layer Processing: The performance advantage of ILP-based composition is shown in Table IV, which measures the benefit of integrating checksumming and byteswapping routines into the memory transfer operation. This experiment compares two data manipulation strategies for two operations: copy with checksum, and copy with checksum and byteswap. The first strategy is nonintegrated processing, or *separate*, representing the case where data arrives and is copied, then checksummed, then possibly byteswapped. We show two varieties of this experiment. The *uncached* case represents what happens if much time occurs in between the various data manipulation operations, and the message gets flushed from the cache. The second data manipulation strategy explored is integrated processing. The *C integrated* case represents hand-integrated loops written in C. The final case is dynamic ILP, using just the checksum pipe of Fig. 2 for *copy & checksum* and the composition of the checksum pipe and a byte swapping pipe, composed as shown in Fig. 1 for *copy & checksum & byteswap*.

Even when compared to the *separate* case which does not have a cache flush between the data manipulation operations, integration provides a factor of 1.4 performance benefit, and is clearly worthwhile. In the case where there is a flush, integration provides a factor of 1.6 performance improvement. The table also demonstrates that our emitted copying routines are very close in efficiency to carefully hand-optimized integrated loops.

TABLE IV
THROUGHPUT OF INTEGRATED AND NONINTEGRATED MEMORY OPERATIONS, MEASURED IN MEGABYTES PER SECOND

Method	copy & checksum	copy & checksum & byteswap
Separate	11	5.8
Separate/uncached	10	5.1
C integrated	16	8.3
DILP	17	8.2

B. Low-Latency Data Transfer

The need for low-latency data transfer pervades distributed systems. The use of ASHs allows applications to quickly respond to messages without paying the higher cost of application upcalls.

In Table V we measure the effects of ASHs and upcalls on raw roundtrip times for a simple remote increment message. When a message arrives, the application (either at user level or in an ASH or upcall handler) receives the message, performs an increment, then responds with another message.

We consider two cases: when the message arrives at the processor the application process is nearly always running [*Currently running* (*polling*)], and when the message arrives at the processor the application process is nearly never running [*Suspended* (*interrupts*)]. In the first case, the application is scheduled, and is actively polling for the message at its time of arrival. In the second case, the application is not scheduled, but is rescheduled as soon as the message reaches user level.[3] If the ASH or upcall completely handles the message, there is no rescheduling of the application.

The use of the ASH saves a significant amount of time (30 μs) as compared to the user-level versions, even when compared to the polling version. When the process is not running, the difference is even more dramatic (96 μs), because the application does not have to be rescheduled in order to run the ASH.

The upcall time is slower than both the ASH time and the user-level polling version time. This is for two reasons: 1) because the upcall mechanism was designed to batch messages together to avoid multiple (potentially expensive) kernel crossings and 2) because the upcall version was not as aggressively optimized for the special case of when the application process is running at message arrival. As expected, however, when the application is not running, the upcall time hardly increases, and becomes much better than the user-level time.

In addition to demonstrating the ability of ASHs to transfer small amounts of data quickly, this experiment also demonstrates the low cost of control transfer and message initiation in our system. Although sandboxing the ASH added little time in absolute terms to the cost of the ping-pong, 76 instructions

[3]Because Aegis' scheduler is round-robin, we had to simulate the time for an interrupt, i.e., for a message arriving to cause the descheduling of the currently running process and the rescheduling of the one the message was for. This was done by setting up a dummy process which does nothing but poll for incoming messages to the application process. When it discovers a message, it immediately yields to the application process. If the application process was polling for the message, it then will receive it right away. This time should closely approximate that for taking an interrupt on a real system.

TABLE V
RAW ROUNDTRIP TIMES FOR REMOTE INCREMENT (IN MICROSECONDS) MEASURED FOR A SANDBOXED ASH, AN UNSAFE (NOT SANDBOXED) ASH, AN UPCALL AND NORMAL USER-LEVEL COMMUNICATION

Process state	Unsafe ASH	Sandboxed ASH	Upcall	User-level
Currently running (polling)	147	152	191	182
Suspended (interrupts)	147	151	193	247

TABLE VI
THE LATENCY (IN MICROSECONDS) AND THROUGHPUT (IN MEGABYTES PER SECOND) FOR TCP RUNNING ON THE AN2 FOR A VARIETY OF CASES

Measurement	Sandboxed ASH	Unsafe ASH	Upcall	User-level (interrupt)	User-level (polling)
Latency	394	348	382	459	384
Throughput	4.32	4.53	4.27	3.92	4.11
Throughput (small MSS)	2.66	3.05	2.78	2.32	2.56

were added to the dynamic instruction base count of 90 for processing this message. We expect this number to decrease somewhat as our prototype sandboxer improves.

Our TCP implementation lowers the cost of data transfer by placing the common-case fast path in a handler which can be run either as an ASH or an upcall. This handler employs dynamic ILP to combine the checksum and copy of message data. A handler can run when the following constraints are satisfied: the packet is "expected" (the packet we receive is the one we have predicted), the user-level TCP library is not currently using that Transmission Control Block (to avoid concurrency problems between the library and the handler), and the TCP library is not behind in processing, so that messages stay in order. If these constraints are violated, the handler aborts and the message is handled by the user-level library. When the header prediction constraint is met, the handler nearly never needs to abort for the other reasons (non-header-prediction-related aborts occurred less than 0.2% of the time in our latency and throughput experiments).

As shown in Table VI, the use of sandboxed ASHs enables a 65 μs improvement in latency over the case of normal user-level TCP when the applications in question are not scheduled at the time of message arrival. When the applications are scheduled, and are doing nothing but polling, sandboxed ASHs are about 10 μs slower. We expect that the performance of our sandboxer can be improved. Nevertheless, the polling version is unrealistic; processes which are part of a multiprogrammed workload yet poll cause poor performance for the system as a whole, therefore we expect the 65 μs difference to be more typical. The upcall performance is slightly better than that of the user-level application polling.

For this latency experiment, there is a limit on the performance that the handler versions can achieve. Because of the way the experiment is set up, the application is responsible for performing each write; the handler only takes care of the read (i.e., placing the data in the right place). This effect occurs because this version of the TCP library implements ASHs completely transparently to applications. An application writer who wished to take full advantage of the power available to handlers while using TCP would have to be cognizant of the use of ASHs, and download an ASH which built on the TCP one.

We show two throughput experiments. The first one uses the same parameters as the one of Table II, and shows that the use of dynamic ILP in sandboxed ASHs enables a 0.4-Mbyte/s gain in throughput when the applications are not scheduled, and a 0.2-Mbyte/s gain in throughput when the applications are scheduled, providing performance similar to that of the *in place, with checksum* experiment of Table II. The upcall version also benefits from DILP, achieving a 0.16-Mbyte/s gain in throughput over the polling version.

For the second throughput experiment, we decreased the Maximum Segment Size (MSS) that the TCP library used. The MSS controls the largest size chunks of data that a TCP implementation sends; if an application requests a send of a larger amount of data, the library transparently communicates multiple times until it has sent all of the data. Although a larger MSS (up to the size of the maximum buffer size of the underlying network) is often better, especially for local communication, the default nonlocal size is normally only 536 bytes [45]. We therefore performed a throughput experiment with the MSS set to this size; this is advantageous to handlers, for there is more work to be done which is application-independent and can thus be handled by the library ASH transparently to the application. For this experiment, we also decreased the size of the buffers being sent to 4096 bytes (from 8192); this decrease is disadvantageous to handlers, because it decreases the amount of application-independent work to be done. As shown by Table VI, when a smaller MSS is being used, even with smaller data sizes being sent by the application, the benefits that handlers bring to applications are increased, in this case by approximately a factor of two.

C. Control Transfer

Low-latency *control* transfer is also crucial to the performance of tightly coupled distributed systems. Examples include remote lock acquisition, reference counting, voting, global barriers, object location queries, and method invocations. The need for low-latency remote computation is so overwhelming that the parallel community has spawned a new paradigm of programming built around the concept of active messages [53]: an efficient, unprotected transfer of control to the application in the interrupt handler.

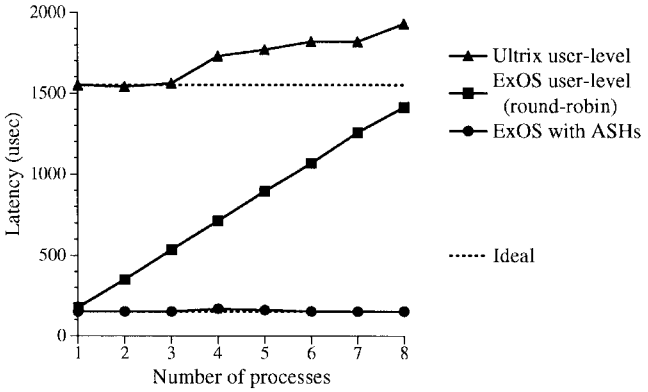

Fig. 4. As the number of processes on the system increases, the cost of waiting for a process to be scheduled becomes increasingly higher; times are in microseconds per round trip.

A key benefit of both ASHs and upcalls is that because the runtime of downloaded code is bounded, they can be run in situations when performing a full context switch to an unscheduled application is impractical. Handlers thus allow applications to decouple latency-critical operations such as message reply from process scheduling. Past systems precluded protected, low-latency control transfer, or heavily relied on user-level polling to achieve performance (e.g., in U-Net using signals to indicate the arrival of a message instead of polling adds 60 μs to the 65-μs roundtrip latency [52]). The cost of control transfers is sufficiently high that recently a dichotomy has been drawn between control and data transfer in the interests of constructing systems to efficiently perform just data transfer [48]. Handlers remove the restrictive cost of control transfer for those operations that can be expressed in terms of handlers. We believe that the expressiveness of handlers as we have described them in this paper is sufficient for most operations subject to low-latency requirements.

As a simple experiment to illustrate the advantages of decoupling latency-critical operations from scheduling a process, we compare executing code in an in-kernel ASH versus in a user-level process while increasing the number of user processes on the client. We consider two user-level cases. In the first, the scheduler is oblivious to message arrival and thus a process with a message waiting for it will not see the message until its turn to run. This case was measured on Aegis, and the processes were scheduled in a round-robin fashion. The second case is with a scheduler that does reschedule a process with a message waiting for it; this set of measurements was taken under Ultrix.

As shown in Fig. 4, as the number of active processes under an oblivious scheduling policy increases, the latency for the roundtrip remote increment increases, because the scheduler is not integrated with the communication system, and does not know to increase the priority of a process that has a message waiting for it. When ASHs are used, on the other hand, the roundtrip time for the remote increment stays much closer to constant, despite the increase in the number of processes. Ultrix uses a more sophisticated scheduler that raises the priority of a process immediately after a network interrupt. As Fig. 4 shows, this type of scheduler definitely reduces the measured effect, but it is certainly still a problem.

Even when the destination process is running and polling the network, ASHs can still provide benefit. The user-level raw latency experiment was performed under scheduling conditions highly favorable to the application: there was only one user process running (the application sitting in a tight poll loop). As shown by the remote increment experiments of Table V, the use of ASHs still provided great benefit, eliminating the system call overhead, the cost of the full context switch to the application, and several writes to the AN2 board. The use of an upcall also eliminates the cost of the full context switch, but does not allow the elimination of system call overhead (nor of the writes to the AN2 board, as long as the upcalls are batched).

D. Sandboxing Overhead

We are interested in measuring both the gains achievable by writing completely application-specific code and how the overhead of sandboxing can cut into these gains. Therefore, we compare the execution time for a generic untrusted remote write (such as might be implemented in the kernel by hand) to that for a sandboxed application-specific remote write (such as might be implemented by an application writer, and downloaded into a kernel). We take this measurement in isolation, without the cost of communication, but with both ASHs running in the kernel.[4] The remote write, modeled after that of Thekkath *et al.* [48], reads the segment number, offset, and size from the message, uses address translation tables to determine the correct place to write the data to, and then writes the data (assuming the request is valid). The application-specific version not only assumes the message was sent by a trusted sender, but also uses a different protocol for communication: the handler assumes it is given a pointer to memory, instead of a segment descriptor and offset. This protocol would clearly not be applicable for all applications, but those that could benefit by it (such as a distributed shared memory system comprised of trusted threads) should not be forced into a more expensive model.

We measured the time for the sandboxed version of trusted ASHs to be 1.3–1.4 times as long as the time for the non-sandboxed for 40-byte writes; for 4096 bytes this factor dropped to 1.01–1.02 times. We emphasize that these overheads are for a completely untuned implementation of sandboxing. An examination of the generated code shows that a large fraction of the added instructions are due to overly general exit code, which could relatively easily be removed, thereby reducing the sandboxing overhead in this case to an unimportant fraction of the runtime.

The dynamic instruction count (excluding data copying) for the application-specific version uses 38 instructions, 28 of which are added by the sandboxer (i.e., the hand-crafted version takes only ten instructions). This shows that when performing the same operation, ASHs are very close in performance to hand-crafted routines. Furthermore, since ASHs can use application-specific knowledge, they can be implemented *more* efficiently than inflexible kernel routines. For example, because it can exploit application semantics (i.e., an organization of trusted peers in a distributed shared memory

[4]With the cost of communication included, the sandboxing overhead disappears in the noise.

system), even the sandboxed version of the specialized remote write (38 instructions) uses fewer instructions than the generic hand-crafted one (68 instructions).

E. Discussion

For our particular implementation, the overhead of sandboxing significantly cuts into the gains achieved by ASHs. Nevertheless, we are optimistic about the role of ASHs for several reasons. First, as we have noted before, the costs of kernel crossings for our system are much cheaper than in normal operating systems, so ASHs should matter more for other implementations. Second, our sandboxer has been optimized for correctness rather than for performance, and should be able to be improved. Finally, different techniques can be used to ensure that ASHs are safe for different systems. For example, the Intel x86 provides hardware support which obviates the need to check loads and stores at runtime. Even on systems without special-purpose hardware, techniques such as *proof-carrying code* [37] may provide the ability to download complex kernel code with no extra runtime overhead.

F. Complexity of the System

The support required to implement static ASHs for our platform is about 1000 lines of code (mostly C, approximately 50 in assembly language) mostly added to the kernel plus 3300 lines of C++ code for the sandboxer. The bulk of the code added to the kernel is to keep track of the ASHs belonging to each application. The additional support required for upcalls is about 400 lines (about 200 in the kernel and the rest in the library operating system). The upcall code leveraged the existing interrupt code a great deal (i.e., no new code had to be added to register or enable interrupt handlers).

We believe that adding efficient upcalls to a standard operating system may be more difficult than adding ASHs (because ASHs involve less operating system mechanism), but have not performed this analysis. Liedtke, a researcher who has implemented highly efficient upcalls, believes that the only way to do so is to completely rewrite the operating system from scratch [30].

The handler code written for ASHs and upcalls themselves tends to be simpler than general-purpose applications, once a programmer figures out which are the important cases that should be handled in an handler. On the other hand, adding handlers to an application may introduce issues of asynchrony for the first time.

In order to implement dynamic ILP, we added 250 lines of interface code plus the VCODE system. VCODE is an independent, released code package of about 3000 lines of code; its use is amortized over multiple OS functions in our environment (including dynamic packet filters [19]). Given that VCODE is a stand-alone package, we find providing DILP to be worthwhile, as DILP greatly simplifies writing efficient integrated loops.

VI. Related Work

The work related to this paper falls into four classes.

1) Computational Model: ASHs can been viewed as a restricted form of Clark's upcalls [7]. Upcalls eliminate layer crossing overhead by synchronously transferring control from a lower level to a higher level (and so forth until a sink is reached) via procedure call rather than via threads. Clark proposed upcalls in the context of an operating system where all code was trusted. We extend his concept by integrating application code (in the form of ASHs) into this "fast path," thereby removing the cost of passing control from the OS to the user process.

This paper also examined fast asynchronous upcalls from OS to application, which differ from the upcalls (or *exceptions*) of modern operating systems by allowing control to be transferred to a process that is not running. Our upcalls are similar to those of Liedtke [30], which obtain efficiency by performing an address space switch rather than a full context switch. It is this type of upcall that we compare ASHs to in this paper.

ASHs trade expressivity for speed by restricting allowed actions in exchange for the elimination of kernel/user boundary crossings. Upcalls make the opposite trade-off. As such the techniques are complementary: simple, small-latency operations can be downloaded as an ASH while longer running computations can perform upcalls to obtain a more expressive environment.

2) Safe Code Importation: There are a number of antecedents to our work on making ASHs safe: Deutsch's seminal paper [11] and Wahbe *et al.*'s modern revisitation of safe code importation [54] influenced our ideas strongly. This work can be viewed as a natural extension of the same philosophical foundation that inspired the packet filter [35]: we have provided a framework that allows applications outside of the operating system to install new functionality without kernel modifications.

The SPIN project [4] is concurrently investigating the use of downloading code into the kernel. SPIN's Plexus network system runs user code fragments in the interrupt handler [22] or as a kernel thread. Plexus guarantees safety by requiring that these code fragments are written in a type-safe language, Modula-3. Plexus simplifies protocol composition, but unlike ASHs, does not provide direct support for dynamic ILP. Preliminary Plexus numbers for in-kernel UDP on Ethernet and ATM look promising but are slower than our *user-level* implementation of UDP. No numbers are reported yet for TCP.

With the advent of HotJava and Java [23], code importation in the form of mobile code has received a lot of press. Recently Tennenhouse and Wetherall have proposed to use mobile code to build Active Networks [46]; in an active network, protocols are replaced by programs, which are safely executed on routers on message arrival. Small and Seltzer compare a number of approaches to safely executing untrusted code [42].

The Vino project is also investigating means for safely importing code into the kernel [41]. They prevent application misbehavior using a heavyweight transactional model; in contrast, we obtain simple safety by restricting the interface ASHs can use. Additionally, the Vino sandboxer generates very inefficient code when large amounts of data manipulation must be done because each reference is sandboxed; dynamic ILP lets us avoid this overhead for data manipulations.

Necula and Lee are investigating Proof Carrying Code, a method where code carries with it a proof that it is safe [37].

These proofs can be used to eliminate some runtime checks. While they have only been able to automate proof generation for very small examples, we regard their work as a promising direction towards making ASHs faster.

3) ASH Benefits: The particular abilities that ASHs provide have been provided in part by other networking systems, though not all together.

a) Message Vectoring: Message vectoring has been a popular focus of the networking community [14]–[16], [39], [52]. The main difference between our work and previous work is that ASHs can perform application-specific computation at message arrival. By using application state and domain knowledge these handlers can perform operations difficult in the context of static protocol specifications.

Application Device Channels (ADCs) [15] are a different approach to eliminating protection domain boundaries from the common communication path. In this approach, much of the network device driver is linked with the application, and many messages can be handled without OS intervention. We would expect this approach to be slower than ASHs when the application is not running, because the entire application/driver process must be scheduled to handle the message.

The most similar work to the ASH system is Edwards *et al.* [16], [17], who import simple scripts using the Unix ioctl system call to copy messages to their destination. The main differences are the expressiveness of the two implementations. Their system supplies only rudimentary operations (e.g., copy and allocate), and provides no way for applications to transfer control, reply to messages or synthesize data operations (such as checksumming and encryption). Nevertheless, their simple interface is easy to implement and tune; it remains to be seen if the expressiveness we provide is superior to it for real applications on real systems.

b) Control Initiation: In the parallel community the concept of *active messages* [53] has become popular, since it dramatically decreases latency by executing code directly in the message handler. Active messages on parallel machines do not worry about issues of software protection.

Several user-level AM implementations for networks of workstations have recently become available. U-Net provides protection, but only at a cost of higher latency: messages are not executed until the corresponding process happens to be scheduled by the kernel [52]. HPAM makes the optimistic assumption that incoming messages are intended for the currently running process; messages intended for other processes are copied multiple times [32]. In contrast, our work can be viewed as an extension of active messages to a general purpose environment that preserves small latencies while also providing protection.

c) ILP and Protocol Composition: There have been many instances of *ad hoc* ILP, for example, in many networking kernels [8]. There is also quite a bit of work on protocol composition [5], [24], [25], [50], [51].

The first system to provide an automatic modular mechanism for ILP is Abbott and Peterson [1]. They describe an ILP system that composes macros into integrated loops at compile time, eliminating multiple data traversals. They provide a thorough exploration of the issues in ILP: most of their analysis can be applied directly to our system. There are two main differences between our system and what they describe: their system is intended for static composition, whereas our system allows dynamic composition, and they make no provisions for application extensions, whereas our system allows untrusted code to participate in ILP in a safe and efficient manner. Proebsting and Watterson describe a new algorithm for static ILP using filter fusion [38].

Static composition requires that all desired compositions be known and performed at compilation time. There are two main drawbacks to such an approach. The first is the exponential code growth inherent in it since the system must pre-compose all *possible* data manipulations rather than those that are used. Furthermore, if dynamic protocol composition is to be supported all protocol data steps must be added, making the order of code growth factorial. In contrast, dynamic composition allows these operations to be combined as need be, scaling memory consumption linearly in proportion to actual use. The second, more subtle problem of static composition is that the system is a closed one: the operating system can neither extend the ILP processing it performs nor have it extended by applications. In contrast, the ASH system allows new manipulation functions to be dynamically incorporated into the system.

4) Scheduling: The correct addition of ASHs to an operating system which has no receive livelock under a constant stream of network interrupts [34] will not reintroduce the problem. To avoid livelock, the operating system must track the number of ASHs recently executed for each process and refuse to execute any more for processes receiving more than their share of messages.

A similar approach to fairly and stably dealing with high communication loads is described in Druschel and Banga [13]. The two key techniques described in their paper, lazy protocol processing at the receiver's priority and early demultiplexing, are both used in the Aegis operating system that our work was performed on to provide fairness and stability under high load. ASHs are fundamentally an eager, not a lazy technique; using them when the system is under a light-to-medium load and then disabling them when the system is under a high load would result in a system having the benefits both of ASHs and of fairness and stability under high load conditions.

Patrick Sobalvarro has explored *demand-based coscheduling*, where processes that are communicating with one another on different machines are scheduled simultaneously [44], increasing the likelihood that messages sent within the group will arrive when their receiver is scheduled. This idea should be useful for particular types of applications, such as parallel applications, but not for ones with less predictable communication patterns (such as a web server).

VII. Conclusion

We have described an extensible, efficient networking subsystem that provides two important facilities: the ability to safely incorporate untrusted application-specific handlers into the networking system, and the dynamic, modular composition of data manipulation steps into an integrated, efficient data

transfer engine. Taken in tandem, these two abilities enable a general-purpose, modular, and efficient method of simultaneously providing both high-throughput and low-latency communication. Furthermore, since application code directs all operations, designers can exploit application-specific knowledge and semantics to improve efficiency beyond that attainable by fixed, hard-coded implementations. Although we have presented results in this paper mainly for the Internet protocols, we have also found ASHs useful in another context: that of executing the software distributed shared memory actions of CRL [26] for various parallel applications; we fully expect them to be useful in other contexts as well.

ACKNOWLEDGMENT

The authors would like to thank the DEC Systems Research Center for lending them four AN2 Network Interface cards and an AN2 switch, and for giving them device drivers for these cards. They are especially grateful to M. Burrows, for providing useful and speedy assistance while they were porting the device drivers to Aegis, and H. Murray, for helping them get the switch running. Thanks are also due to T. Pinckney and H. M. Briceño for implementing large parts of the support software for interacting with Aegis, and to E. Kohler, for writing the sandboxer and also for providing extensive and insightful comments on a draft of this paper.

REFERENCES

[1] M. B. Abbott and L. L. Peterson, "Increasing network throughput by integrating protocol layers," *IEEE/ACM Trans. Networking*, vol. 1, pp. 600–610, Oct. 1993.

[2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 109–126.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PATHFINDER: A pattern-based packet classifier," in *Proc. First Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994, pp. 115–123.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety and performance in the SPIN operating system," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 267–284.

[5] N. T. Bhatti and R. D. Schlichting, "A system for constructing configurable high-level protocols," in *ACM SIGCOMM'95*, Cambridge, MA, Aug. 1995, pp. 138–150.

[6] R. Braden, D. Borman, and C. Partridge, *Computing the Internet Checksum*. RFC 1071.

[7] D. D. Clark, "The structuring of systems using upcalls," in *Proc. 10th Symp. on Operating Systems Principles*, Orcas Island, WA, Dec. 1985, pp. 171–180.

[8] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Commun. Mag.*, vol. 27, no. 6, pp. 23–29, June 1989.

[9] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, Philadelphia, PA, Sept. 1990, pp. 200–208.

[10] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Supercomputing*, Portland, OR, Nov. 1993, pp. 262–273.

[11] P. Deutsch and C. A. Grant, "A flexible measurement tool for software systems," *Inform. Processing 71*, 1971.

[12] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson, "Network subsystem design," *IEEE Network*, vol. 7, no. 4, pp. 8–17, July 1993.

[13] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, pp. 261–275.

[14] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proc. 14th Symp. on Operating Systems Principles*, Asheville, NC, Dec. 1993, pp. 175–189.

[15] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, London, U.K., Aug. 1994, pp. 2–13.

[16] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, London, U.K., Aug. 1994, pp. 14–24.

[17] A. Edwards and S. Muir, "Experiences implementing a high performance TCP in user-space," in *Conf. Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM'95)*, Cambridge, MA, Aug. 1995, pp. 196–205.

[18] D. R. Engler, "VCODE: A retargetable, extensible, very fast dynamic code generation system," in *Proc. SIGPLAN'96 Conf. on Programming Language Design and Implementation*, Philadelphia, PA, May 1996, pp. 160–170.

[19] D. R. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM'96)*, Stanford, CA, Aug. 1996, pp. 53–59.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-specific resource management," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 251–266.

[21] D. R. Engler, D. A. Wallach, and M. F. Kaashoek, "Design and implementation of a modular, flexible, and fast system for dynamic protocol composition," M.I.T. Lab. Computer Science, Tech. Memo. TM-552, May 1996.

[22] M. E. Fiuczynski and B. N. Bershad, "An extensible protocol architecture for application-specific networking," in *Proc. USENIX*, San Diego, CA, Jan. 1996, pp. 55–64.

[23] J. Gosling, "Java intermediate bytecodes," in *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, San Francisco, CA, Mar. 1995, pp. 111–118.

[24] R. Harper and P. Lee, "Advanced languages for systems software: The Fox project in 1994," Carnegie Mellon Univ., PA, Tech. Rep. CMU-SC-94-104, Jan. 1994.

[25] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Trans. Software Eng.*, vol. 17, pp. 64–76, Jan. 1991.

[26] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-performance all-software distributed shared memory," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 213–228.

[27] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "TreadMarks: Distributed shared memory on standard workstations and operating systems," in *Proc. 1994 Winter USENIX Conf.*, San Francisco, CA, Jan. 1994, pp. 115–132.

[28] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Reading, MA: Addison-Wesley, 1989.

[29] K. Li. "IVY: A shared virtual memory system for parallel computing," in *Int. Conf. on Parallel Computing*, University Park, PA, Aug. 1988, pp. 94–101.

[30] J. Liedtke, "On μ-kernel construction," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, Dec. 1995, pp. 237–250.

[31] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," in *Proc. 14th Symp. on Operating Systems Principles*, Asheville, NC, Dec. 1993, pp. 244–255.

[32] R. P. Martin, "HPAM: An Active Message layer for a network of HP workstations," in *Proc. Hot Interconnects II*, Aug. 1994.

[33] H. Massalin, "Synthesis: An efficient implementation of fundamental operating system services," Ph.D. dissertation, Columbia University, New York, 1992.

[34] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," Digital Western Res. Lab., Tech. Rep. 95/8, Dec. 1995.

[35] J. C. Mogul, R. F. Rashid, and M. J. Accetta," The packet filter: An efficient mechanism for user-level network code," in *Proc. 11th Symp.*

on Operating Systems Principles, Austin, TX, Nov. 1987, pp. 39–51.
[36] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM'96)*, Stanford, CA, Aug. 1996, pp. 73–84.
[37] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, pp. 229–243.
[38] T. A. Proebsting and S. A. Watterson, "Filter fusion," in *Proc. 23rd Annu. Symp. on Principles of Programming Languages*, St. Petersburg, FL, Jan. 1996, pp. 119–130.
[39] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, "High-performance local area communication with fast sockets," in *Proc. USENIX 1997 Annu. Tech. Conf.*, Anaheim, CA, Jan. 1997, pp. 257–274.
[40] D. J. Scales, M. Burrows, and C. A. Thekkath, "Experience with parallel computing on the AN2 network," in *International Parallel Processing Symp.*, Honolulu, HI, Apr. 1996, pp. 94–103.
[41] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *Proc. Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996, pp. 213–227.
[42] C. Small and M. Seltzer, "A comparison of OS extension technologies," in *Proc. USENIX*, San Diego, CA, Jan. 1996, pp. 41–54.
[43] J. M. Smith and C. B. S. Traw, "Giving applications access to Gb/s networking," *IEEE Network*, vol. 7, no. 4, pp. 44–52, July 1993.
[44] P. G. Sobalvarro, "Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Jan. 1997.
[45] R. W. Stevens, "*TCP/IP Illustrated: The Protocols.* Reading, MA: Addison-Wesley, 1994, vol. 1, ch. 18, p. 237.
[46] D. L. Tennenhouse and D. J. Wetherall, "Toward an active network architecture," in *Proc. Multimedia, Computing, and Networking 96*, Jan. 1996.
[47] C. A. Thekkath and H. M. Levy, "Limits to low-latency communication on high-speed networks," *ACM Trans. Computer Syst.* vol. 11, no. 2, pp. 179–203, May 1993.
[48] C. A. Thekkath, H. M. Levy, and E. D. Lazowska, "Separating data and control transfer in distributed operating systems," in *Sixth Int. Conf. on Architecture Support for Programming Languages and Operating Systems*, San Francisco, CA, Oct. 1994, pp. 2–11.
[49] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1993*, San Francisco, CA, Oct. 1993, pp. 64–73.
[50] C. Tschudin, "Flexible protocol stacks," in *Proc. SIGCOMM 1991*, Zurich, Switzerland, Sept. 1991, pp. 197–204.
[51] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. Karr, "A framework for protocol composition in Horus," in *Proc. 14th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, Ottawa, Ont., Canada, Aug. 1995, pp. 138–150.
[52] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proc. 15th Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, 1995, pp. 40–53.
[53] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proc. 19th Int. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, pp. 256–266.
[54] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. "Efficient software-based fault isolation," in *Proc. 14th Symp. on Operating Systems Principles*, Asheville, NC, Dec. 1993, pp. 203–216.
[55] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Floyd, "Implementing real time packet forwarding policies using Streams," in *Proc. USENIX Winter 1995 Tech. Conf.*, New Orleans, LA, Jan. 1995, pp. 71–82.
[56] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "ASHs: Application-specific handlers for high-performance messaging," in *ACM Communication Architectures, Protocols, and Applications (SIGCOMM'96)*, Stanford, CA, Aug. 1996, pp. 40–52.
[57] M. Yuhara, B. Bershad, C. Maeda, and E. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proc. Winter 1994 USENIX Conf.*, San Francisco, CA, Jan. 1994, pp. 153–165.

Deborah A. Wallach received the S.B., S.M., and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge.

She is a member of the research staff in the Western Research Laboratory at Digital Equipment Corporation. Her research interests include distributed systems, operating systems, and networks.

Dawson R. Engler received the B.A. degree from the University of Arizona and the S.M. degree from the Massachusetts Institute of Technology, Cambridge, where he is currently a graduate student.

His research interests are computer systems, primarily operating systems and networking, and compilation, primarily dynamic code generation and application-specific compilation.

M. Frans Kaashoek (S'88–M'89) received the Ph.D. degree from the Vrije Universiteit, Amsterdam, The Netherlands.

Currently, he is a Jamieson Career Development Associate Professor in the Massachusetts Institute of Technology EECS Department, Cambridge, and a Member of the M.I.T. Laboratory for Computer Science. His principal field of interest is computer systems: operating systems, networking, programming languages, compilers, and computer architecture for distributed systems, mobile systems, and parallel systems.