To appear in SIGCOMM '96, August 1996.

ASHs: Application-Specific Handlers for High-Performance Messaging

Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.

{kerr, engler, kaashoek}@lcs.mit.edu

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 provide this high degree of flexibility while still providing network performance as good as, or (if they exploit application-specific knowledge) even better than, hard-wired in-kernel implementations. A combination of user-level microbenchmarks and end-to-end system measurements using TCP demonstrate the benefits of the ASH system.

1 Introduction

Applications' complexity and ambition scale with increases in processing power and network performance. For example, the last few years have seen a proliferation of distributed shared memory systems [25, 26, 28], real-time video and voice applications [45], parallel applications [11, 34], and tightly-coupled distributed systems [2, 36, 39]. 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 [10, 13, 39]. 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 upcalls [8] that are safely and efficiently executed in the kernel in response to a message arrival. ASHs direct message transfers (thereby eliminating copies), incorporate data 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 high-performance implementations of UDP and TCP libraries without ASHs.

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.

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 whom the message is destined). The most important property of ASHs is that they represent bounded, *safe* upcalls. 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 the kernel of 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:

Direct, dynamic message vectoring An ASH dynamically controls where messages are copied in memory, and can therefore eliminate intermediate copies. Because most systems do not allow application-directed message transfers, messages are copied into at least one intermediate buffer before being placed in their final destination.

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, the single most 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.

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 [43] or remote lock acquisition in a distributed shared memory system. Even recently, low-overhead control transfer had been considered to be infeasible to implement [39].

We have also integrated support for *dynamic integrated layer processing* and *dynamic protocol composition* into the ASH system.

Dynamic integrated layer processing (ILP) Current systems often have many 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 costly memory traversals, stressing a weak link in high-performance networking: the memory subsystems of the endpoint nodes. As argued by Clark and Tennenhouse [10], an *integrated* approach, where these operations are combined into a single memory traversal, can greatly improve the latency and throughput of a system.

ASHs integrate data manipulations such as checksumming or conversions into the data transfer engine itself. ASHs automatically and dynamically perform integrated layering processing (ILP). Despite the fact that ASHs improve flexibility by using protocol layers integrated at runtime, dynamic ILP is as efficient as statically-written hard-wired ILP implementations.

Dynamic protocol composition Protocols are the units of modularity in networking. To provide higher level functionality than is provided by any single protocol, they are frequently composed together into protocol stacks. The ASH system provides a simple interface to dynamically compose protocols. Using on-the-fly code generation, these composed protocols are integrated into an efficient data path.

The ASH system has been implemented in an exokernel operating system [19]. Aegis, an exokernel for MIPS-based DECstations, securely exports two network devices: a 10 Mbit/s Ethernet and a 155 Mbit/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. For example, the use of message initiation and control initiation in ASHs improves raw user-level roundtrip latency of an already highly-tuned AN2 ATM network from 176 microseconds to 142 microseconds, independent of whether the application is scheduled or not. In general, we would expect even better performance improvements for using ASHs in other kernels than exokernels, since Aegis has been highly optimized for fast kernel crossings [19], reducing the benefit of downloading code into the kernel.

The remainder of the paper is structured as follows. Section 2 discusses design issues and Section 3 implementation issues for ASHs. Section 4 describes the experimental environment for evaluating the benefits of ASHs. Section 5 reports on how ASHs can be used and illustrates their benefits. Section 6 relates ASHs to other work. In Section 7 we draw our conclusions.

2 Application-specific safe handlers

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. Second, these routines are downloaded into the operating system in the form of machine code, and given 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, then hands an identifier back to the user. Third, the ASH is associated 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 to, 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. ASHs can be completely static, in which case they cannot take advantage of dynamic ILP. This advantage can be gained with the addition of pipes. 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 all three 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. For applications that dynamically compose protocols, ASHs with protocol composition may be used. We describe each of the types of ASHs in turn.

2.1 Static ASHs

Static as well as non-static 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). If there are several actions to be taken here, such as a checksum and a copy, integrated layer processing as described below can occur 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 something goes wrong (for example a checksum fails), the abort code is called to fix up any state that has been updated. If the first two parts completed successfully, on the other hand, 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.

Static ASHs are responsible for hand-orchestrating any data manipulations they require.

2.2 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 traversals 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 [10] 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. Given the richness of possi-

```
// Initialize a pipelist for two pipes
pl = pipel(2);

// Create checksum pipe
checksum_pipe_id = mk_cksum_pipe(pl, &pipe_cksum);
// Create byteswap pipe
byteswap_pipe_id = mk_byteswap_pipe(pl);

// Compile the two pipes,
// returning a handle to the integrated function
ilp = compile_pl(pl, PIPE_WRITE);
```

Figure 1: Compose and compile checksum and byteswap pipes.

ble operations, such mismatches happen quite easily. Furthermore, many systems compose protocols at runtime [5, 23, 24, 41, 42], 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.

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 byte swap, or an accumulation for a checksum). Our ASH pipe compiler can integrate 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 bits, 32 bits, 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-bit words, while an encryption pipe may require 32-bit words. To allow a 16-bit checksum pipe's output to be streamed through a 32-bit encryption pipe, it is aggregated into a single register.

Figure 1 presents an example composition of the checksum pipe of Figure 2 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, which can be handled as an inviolable object.

The pipes for ASHs are written in VCODE [17], which is a low-level extension language 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 Figure 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 are given control over register class. The two 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 (`pipe_cksum` in Figure 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.

Current compilers do not optimize networking idioms well. This is mainly due to the fact that there is no clear idiomatic way 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, we have extended the VCODE system (to which the pipe language is compiled to) to include common networking operations. Importantly, the VCODE system is low-level enough that further extensions can be done by clients of the ASH system with little performance impact.

Figure 2 exploits the extension added 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-bit 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 it. 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 bits.

2.3 ASHs with dynamic protocol composition

In addition to dynamic ILP, ASH programmers can also use the dynamic protocol composition extensions provided by the ASH system. 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 layers (only one IP routine has to be written, and can be composed with UDP or TCP).

ASHs written in this style consist of a collection of routines for sending and receiving messages that are dynamically organized into a stack. The layers in the stack are either protocol layer routines (*e.g.,* UDP) or application specific layers (*e.g.,* a WWW server). Each layer is written in C, heavily augmented with library primitives for message manipulation. Although each layer sees a message as a stream, receiving part of a message as input and producing one as output, the layers are each structured similarly to static ASHs. A layer has an initial body that is run when a message is being constructed or consumed and a final body that is run after all bodies on a given path have executed. The main data processing occurs in between the initial and final parts and uses pipes for all data manipulations.

Each layer is provided with a set of message primitives to initiate and consume messages, add headers to and strip headers off of messages, and reserve header space for information not known until an entire message is processed. For example, a UDP receive layer would typically take in a whole UDP message, consume the header, and produce the body of the message as output passed up to the next layer. Because layers consume parts of messages, not all of the data makes it up to the top of the stack, and different ILP data manipulation loops are generated for different parts of the message.

The receive part of each layer is comprised of three procedures: a body procedure that does the initial processing and invokes data manipulation operations, and two handlers for the final body—abort (called if a lower level aborts) and commit (called if all lower levels

```
// Specify a pipe to compute the Internet checksum and return its identifier.  This specification is subsequently converted
// by our system 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_t *r) {
    reg_t reg;
    int pipe_id;

     // This checksum works with 32 bits and is both commutative and non−modifying (i.e., it 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();
    *r = reg;
    return pipe_id;
}
```

Figure 2: Simple checksum pipe example.

succeed). The body procedure can consume the message or, if necessary, defer processing until the message is certified by the low levels. The body procedure can fail, synchronously returning an error code to the level that invoked it via `deliver`; this level is then responsible for handling the failure (potentially by returning an error in turn to the level that invoked it). After the message is processed, either abort or commit is invoked. During the body processing phase, the ASH system tracks which handlers to call by recording the protocol layers invoked. Every time a layer is activated its handlers are (logically) enqueued on a list. When the final phase is initiated, these handlers are called, in FIFO order. A commit handler is called unless: (1) a lower-level protocol failed, or (2) a lower-level commit handler failed, in which case the abort is propagated upwards (by invoking the abort handlers of bodies affected by the failure). Higher level protocol body failures are synchronously propagated down; a level receiving notification of a failure above can choose to abort (propagating the failure down further) or buffer the data for later and succeed (stopping the chain of failure propagation). The send part of a layer (if any) is similarly structured to the receive path, only without an abort handler.

Figure 3 shows the main body for a very simple ASH that implements naive remote writes [39]. The ASH extracts the message destination address and length from the message (using the `consume` library call). It then copies the payload to the destination address, also using the `consume` call. The message is not passed up further. Note that this layer could be the lowest layer of a protocol stack, or could sit higher, on top of a UDP layer, for example. This naive handler, appropriate for use only on highly reliable networks, treats aborts as catastrophes.

A server discussed in Section 5 uses integrated ASHs for sending and receiving. The receive ASH integrates 3 layers: (1) processing the AAL5 trailer; (2) the computations of IP and TCP checksums; and (3) an application-specific operation that checks whether the data requested is in the server cache. These three operations are independently written as three separate layers and then dynamically composed. When a message arrives, it is demultiplexed, the integrated ASH is run, and if the data is in the cache, it is directly sent back to the client by the ASH. The send path is also specified as a series of layers. This organization allows simple application-specific operations to be easily and safely integrated in the messaging system, allowing for high performance and a high degree of flexibility.

3 Implementation

This section describes how we implemented the ASH system. We discuss the support required from the operating system to run ASHs, describe the strategies to deal with ASHs that abort, and outline the methods for making ASHs safe. We conclude this section with implementation caveats.

3.1 The operating system model

The most important task we require from the operating system is to provide address translation. The primary reason for this requirement is that virtual memory greatly eases the task of writing ASHs. For example, it allows the handlers to execute in the addressing context of their associated application, and thus directly manipulate user-level data structures. If address translations were embedded within the handler, such manipulations would be more difficult.

On the MIPS architecture 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 non-resident page, or an illegal memory address, it is aborted.

For efficiency reasons, we can allow addresses to be pre-bound when the ASHs are imported into the kernel. Pre-binding address translations removes the possibility of virtual memory exceptions, but complicates the programming model. Additionally, to ensure correctness, the operating system must track what pages can be accessed from the ASH: if any of these pages are deallocated or have their protection changed, then the corresponding memory operations must be retranslated or the ASH must be disabled. We do not explore this methodology in this paper.

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.

Note that we do not assume that the operating system can field ASH-induced exceptions, that ASHs necessarily have access to floating point hardware, or that hardware timing mechanisms are available. As discussed in Section 3.3, we have designed safety provisions to remove the necessity for these functions.

```
// Simplified ASH to copy packets from the network buffer to their given destination address.
void simple_remote_write_handler(void *ash_data, int nbytes) {
    int len;   char *dst;
    if (!consume(&dst, sizeof(char *), NULL_PIPE))        // Load destination address (first word)
        return ASH_ABORT;                                 // Short message
    if (!consume(&len, sizeof(int), NULL_PIPE))           // Load length (second word)
        return ASH_ABORT;                                 // Short message
    if (nbytes < len+sizeof(int)+sizeof(char *))          // Incomplete packet
        return ASH_ABORT;
    if (!consume(dst, len, NULL_PIPE))                 // Copy len bytes from (message buffer + 8) to (dst).
        return ASH_ABORT;                              // Short or corrupt packet
    return ASH_SUCCESS;                                // Success.
}
```

Figure 3: Example ASH.

3.2 The abort protocol

When an ASH cannot complete execution it must *abort*. There are many possible causes of an abort. First, external events (*e.g.,* a device interrupt). Second, malicious or buggy ASH operations (*e.g.,* divide by zero or excessive execution). Third, an ASH abort request (*e.g.,* an ASH may need to acquire a lock before continuing execution). Finally, an ASH action (*e.g.,* a page-fault can require that the handler be suspended until that page is resident in main memory).

The current system only handles voluntary ASH aborts. The ASH itself is responsible for allocating all the resources it will need before making any permanent changes, as well as fixing up any changes that it has made if it decides to abort at some point.

We plan to provide the ability to suspend and later restart an ASH (still in the kernel). This ability requires that all of an ASH's state be saved, including its live registers and the message that it was processing. This may require transparent message relocation. Since all message accesses will be stylized in our eventual system, relocation of the message should be a straightforward operation.

3.3 Safe execution

For safety, the ASH system must guard against excessive execution time, exceptions, and wild memory references and jumps in ASHs. There are various ways to guarantee safety, depending 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 software techniques. We describe these software techniques here in detail.

The ASH system for the MIPS bounds execution time using a framework inspired by Deutsch [12]. Exceptions are prevented using runtime and static checks (as is done in existing packet-filters [31, 47]). Wild memory references are prevented using a combination of address-space fire-walls and *sandboxing* [44]. Wild jumps are prevented using language support. We examine each technique in further detail below.

Bounding execution time Because we want to allow four-kilobyte 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). This large instruction count allows us to achieve implementation simplicity by overestimating the effects of straight-line code. As a result, our current estimations of execution time are overly pessimistic, but simple to implement:

- The base cost of an ASH and its pipes is computed by simply counting its total number of instructions.
- For every data transfer, we multiply the number of iterations it performs by the base cost of its associated pipe(s).
- The summation of the static base ASH cost and the dynamic cost of its data transfer operations is the total cost of the ASH. (Note that the cost of the pipe is a constant, and so this multiplication can be strength-reduced).
- At runtime, the total cost is computed and then charged to the ASH; if the ASH exceeds its budget during execution, it is aborted.

  The total number of cycles consumed can be used in process scheduling or in deciding whether to defer ASH invocation. On machines with appropriate hardware, cycle counters can be used to accurately count handler execution times.

We are currently extending this framework to be less pessimistic in its execution time estimation.

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. Arithmetic overflow exceptions are prevented by converting all signed arithmetic instructions to unsigned ones (which do not raise overflow exceptions). At download time, we prevent the usage of floating-point instructions and protect against wild jumps (we do not allow indirect jumps). 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 non-resident pages or illegal addresses, the ASH is aborted.

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.

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

On the MIPS architecture, code executing in kernel mode can read and write physical memory directly. To prevent this, we force all non-message loads and stores to have user-level addresses, using the code inspection (*sandboxing*) techniques of Wahbe et al. [44]. Similarly, we mask the lower bits of memory operations to prevent unaligned exceptions. Loads of kernel-level message data are performed only through specialized function calls: in this way the sandboxer knows no memory operations to kernel-level data (*i.e.,* the received message buffer) should occur.

Making sandboxed data copies efficient is difficult. The ASH system therefore requires that messages are accessed through specialized function calls, preventing user pointers into the message. 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.

3.4 Dynamic code generation

Dynamic code generation is the generation of executable code at runtime. We exploit dynamic code generation techniques to efficiently implement dynamic ILP and dynamic protocol composition. The details of the implementation are more fully described in [20].

3.5 Implementation caveats

The current system as measured in the rest of the paper has three limitations. First, we treat ASH aborts in a very simplistic manner: we do not save any state that the ASH was using, other than associating the ASH with the message that it was working on. However, in many cases this is sufficient, since ASHs can be re-initiated. Second we do not yet provide safe execution for the implementation of the ASH system as described; we have a simple sandboxer for an earlier implementation but it is incompatible with this implementation. This affects our performance numbers positively. However, by now sandboxing is well understood and has proven to result in little performance overhead. In addition, a sandboxer for ASHs is more simple and therefore more efficient than a general sandboxer, since ASHs have a restricted instruction set and have restricted access to other parts of the system. Besides these restrictions, all of the operations we discuss are supported: dynamic compilation of ASHs, dynamic integration using pipes for data transfer, initiation of messages and message vectoring, and downloading code into the kernel.

4 Experimental environment

This section reports on the base performance of our system without ASHs. The next section reports on the benefits of using ASHs. Like other systems [15, 16, 29, 36, 40], all the protocols are implemented in user space. The main point to take from the results in this section is that our implementation without ASHs 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.

4.1 Testbed

We have implemented a system for ASHs in an exokernel operating system [19]. Although our implementation is for an exokernel, ASHs are largely independent of the specific operating system and operating system architecture. They 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 Mbit/s Ethernet and a 155 Mbit/s AN2 (Digital's ATM network). The Ethernet device is securely exported by a packet filter engine [31]. The Aegis implementation of the packet filter engine, DPF [18], 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 (MPF [47] and PATHFINDER [3]) in the literature.

Network	**Latency**
in-kernel AN2	112
user-level AN2	176
Ethernet	279

Table 1: Raw latency (in microseconds per round trip) for user-level and in-kernel applications on AN2 and Ethernet.

Similarly to other systems [15, 34, 36], the AN2 device is securely exported by using the ATM connection identifier to demultiplex packets. 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.

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 direct-mapped write-through 64KB caches for instructions and data. Memory and I/O devices are accessed over a 25-Mhz TURBOchannel bus. The two DECstation 240s are connected with an AN2 switch.

While collecting the numbers reported in this paper, we had a fair number of problems with cache conflicts (similar to problems reported by others [32]), because the DECstations have direct-mapped caches. We took two steps in order to minimize the effect these conflicts had on our experiments: first, after examining the results from linking object files in many different orders, we picked a best-case timing to report, and second, for any set of related experiments, such as the user-level UDP family, we included the identical amount of code statically; only the dynamic path through the code changed. We feel that this methodology provided a fair comparison between the different experiments. If anything, it should be disadvantageous towards the ASH measurements, as they should be less likely to suffer from cache conflicts in a real system than the user-level measurements because the code involved is tightly clustered together.

4.2 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 1 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 this configuration, the Ethernet number (279 microseconds) is close to the limits measured for user-level low-latency communication in [38]; an exact comparison is complicated since the limits were measured on DECstation 5000/200s.

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

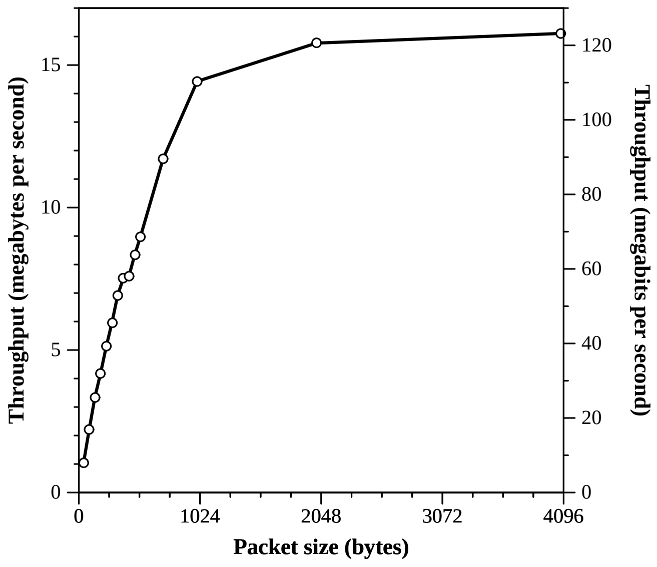

Figure 4: Throughput for a user-level application on the AN2.

hardware overhead for a round trip is approximately 96 microseconds [34], the kernel software is adding only 16 microseconds 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 we designed for the board, adds another 64 microseconds, which brings the total software overhead to 80 microseconds, or about 3,200 cycles.

Figure 4 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/second (134 Mbits/second) [34]. At a 4-Kbyte packet size, we reach 16.11 MBytes/second.

These raw numbers are competitive with other high-performance implementations that also export the network to user space. Scales et al. [34] measure about twice as much software overhead (7,600 cycles or 34 microseconds) for a null packet send using their `pvm_send` and `pvm_receive` interface using the same ATM board, with a substantially faster machine (a 225-MHz DEC 3000 Model 700 AlphaStation rated at 157 SPECint92). Our absolute numbers are higher than U-Net (176 vs 66 microseconds), since our experiments are taken on slower machines (40-Mhz vs. 66-Mhz), the AN2 hardware latency is higher than the Fore latency (96 microseconds vs. 42 microseconds), and we have not attempted to rewrite the AN2 firmware to achieve low latency, as was done for U-Net [36]. Direct comparisons with other high-performance systems such as Osiris [15] and Afterburner [16] are difficult since they run on different networks and have special purpose network cards, but our implementation appears to be competitive.

4.3 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, 36]. 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.

Implementation	**Latency**	**Throughput**
AN2; in place, no checksum	203	11.61
AN2; in place, with checksum	226	7.42
AN2; no checksum	207	8.84
AN2; with checksum	228	6.67
Ethernet; with checksum	309	1.03

Table 2: Latency and throughput for UDP over AN2 and Ethernet. The latency is measured in microseconds, and the throughput in megabytes per second.

Tables 2 and 3 show the latency and throughput for 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 destined 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 true zero-copy can actually be achieved; with our user-level AN2 interface the application can be informed where the data has landed, and can 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. For internetworks, the *no checksum* implementations are clearly inadequate because they does not offer an end-to-end checksum. We thus 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 implementation is closest to what one might expect from an hard-coded in-kernel implementation.

Table 2 shows 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 6 maximum-segment-size packets (1,500 bytes for Ethernet and 3,072 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 process speed differences about the same as the fastest implementation reported in the literature [38]. Using the AN2 interface, UDP latencies are about 31 microseconds 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 [36]; the U-Net implementation adds 73 microseconds on a 66-Mhz processor while our implementation adds 52 microseconds on a 40-Mhz processor (even though, unlike their numbers, our checksum and memory copy are not integrated for this measurement). 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

Implementation	Latency	Throughput
AN2; in place, no checksum	333	6.08
AN2; in place, with checksum	364	4.44
AN2; no checksum	333	5.46
AN2; with checksum	364	4.10
Ethernet; with checksum	412	1.04

Table 3: Latency and throughput for TCP over AN2 and Ethernet. The latency is measured in microseconds, and the throughput in megabytes per second.

bandwidth.

Table 3 shows 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 3,072 bytes and for the Ethernet the maximum segment size is 1,500 bytes. For both networks the window size was fixed at 8 Kbytes. Larger window size increases the throughput. Except during connection set up and tear down, all segments were processed by the header-prediction code.

The difference between UDP and TCP latency is mostly accounted for by 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 non-optimized protocol processing (checking the validity of the segment received and running header-prediction code). The sources of overhead, together accounting for about 130 microseconds, seem also to account for most of the difference in latency with U-Net, which adds a total of 20 microseconds (on a 66-Mhz machine) over their UDP implementation.

In summary, the base performance of our system for UDP and TCP is in the same ballpark or is better than most high-performance user-level and in-kernel implementations [15, 16, 21, 29, 40].

5 Using application-specific handlers

In this section, we examine how application-specific safe handlers can be exploited to achieve good throughput, data transfer latency and control transfer latency. Many of our experiments are influenced by Clark and Tennenhouse [10].

We use a combination of user-level microbenchmarks and end-to-end system measurements. The microbenchmarks gauge the individual effects of, for example, avoiding copies, while the system 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 a 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. Also, as pointed out by Braun et al. [7], it is important to evaluate ILP in a complete protocol environment. For most of our experiments we used dynamic ILP but not protocol composition in order to separate out the cost of dynamically composing ASHs. The end-to-end measurements of the server, however,

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

Table 4: 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.

include the cost for dynamically composing ASHs.

ASHs can be used for a number of purposes. Some require that the ASH is executed in the kernel; for others, ASHs can be run in user space. In the experiments we attempt to separate out the additional benefits of downloading and running an ASH in the kernel. It should be noted that the results of our experiments greatly underestimate the benefits of running ASHs in any other kernel, because kernel crossings in Aegis have been highly optimized: Aegis kernel's crossings 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 [19]. For example, the advantage of running an ASH in the Aegis exokernel versus running an ASH in user space is 34 microseconds; in a system like Ultrix this difference would be more like 130 microseconds (the approximate cost of an exception plus the system call back into the kernel).

5.1 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 [10, 13]. To address this bottleneck, applications must be able to direct message placement, and to exploit ILP during copying. We examine each below.

5.1.1 Avoiding message copies

Message copies cripple networking performance [1, 10, 39]. However, most network systems make little provision for application-directed data transfer. This results in needless data copies as incoming messages are copied from network buffers to intermediate buffers (*e.g.*, BSD's mbufs [27]) and then copied to their eventual destination. To solve this problem, we allow an ASH 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: ASHs 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 ASHs 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 4 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 can observe this effect even in our UDP and TCP implementations: the throughput for the *no checksum* version of UDP increases by a factor of 1.2–1.3 when the copy from the network buffers into to the application's data structures is eliminated.

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

Table 5: Cost of integrated and non-integrated memory operations. Throughput is measured in megabytes per second.

The ASH 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 an ASH, 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. Applications which require the data be copied, on the other hand, can use ASHs 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.

5.1.2 Integrated layer processing

The performance advantage of ILP-based composition is shown in Table 5, 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 non-integrated 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 Figure 2 for *copy & checksum* and the composition of the checksum pipe and a byte swapping pipe, composed as shown in Figure 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.

5.2 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 high cost of application upcalls.

In Table 6 we measure the effects of ASHs on raw roundtrip times for a simple remote increment message. The ASH receives a message, performs an increment, then responds with another message. This experiment also demonstrates the low cost of control transfer and message initiation in our system.

Our TCP implementation allows ASHs to lower the cost of data transfer by downloading the common-case fast path in the kernel.

Network	With ASH	Without ASH
AN2	142	176

Table 6: Raw roundtrip times for remote increment (in microseconds) with and without ASHs.

Measurement	With ASH	Without ASH
Latency	347	364
Throughput	4.52	4.10

Table 7: Latency (microseconds) and throughput (megabytes per second) for TCP on AN2 with and without ASHs.

An ASH 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 TCB (concurrency control), and the TCP library is not behind in processing, so that messages stay in order. If these constraints are not satisfied, the ASH aborts and the message is handled by the user-level library. When the header prediction constraint is met, the ASH nearly never needs to abort for the other reasons (non-header-prediction-related aborts occurred less than 0.5% of the time in our latency and throughput experiments). As shown in Table 7, the use of ASHs enables a 17 microsecond improvement in latency, and a .4 MByte/second gain in throughput, providing performance better than the *in place, with checksum* experiment of Table 3. The improvement in latency occurs despite the fact that no messages are initiated from the TCP ASH during the latency experiment.

To estimate the cost of sandboxing for a complete implementation of the ASH system, we have compared the ASH time for a generic untrusted remote write to that for an application-specific remote write in isolation without the cost of communication, using the simple sandboxer we developed for an earlier implementation of the ASH system (see Section 3.5). The remote write, modeled after that of Thekkath et al. [39], 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 found the overhead of sandboxing for 40-byte writes to be 0–13%, and for 128 bytes 3–7%. We strongly emphasize that these overheads are for a very rudimentary implementation of sandboxing: we sandbox every load and store, rather than only register definitions. An examination of the generated code shows that even simple flow analysis is sufficient to remove almost all sandboxing instructions, thereby reducing sandboxing overhead to an unimportant fraction of runtime.

As this data shows, when performing the same operation, ASHs are very close in performance to hand-crafted routines. Furthermore, since ASHs can utilize 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 system), the specialized remote write facility is approximately 12% faster than the "in-kernel" untrusted remote write facility.

5.3 Control transfer

Low-latency *control* transfer is also crucial to the performance of tightly-coupled distributed systems. Examples include remote lock allocation, 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 [43]: an efficient, unprotected transfer of control to the application in the interrupt handler.

A key benefit of ASHs 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. ASHs 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 30 microseconds to the 65-microsecond roundtrip latency [36]). 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 [39]. ASHs remove the restrictive cost of control transfer for those operations that can be expressed in terms of ASHs. We believe that the expressiveness of ASHs 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. In the experiment, the user-level processes are scheduled round-robin. As shown in Figure 5, as the number of active processes 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 nearly constant, despite the increase in the number of processes. A more sophisticated scheduler that raises the priority of a process immediately on a network interrupt would reduce the measured effect, but is not a general solution either (*e.g.,* if the message is not latency critical).

Even when the destined 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 6, 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.

5.4 Dynamic protocol composition: a simple server

One of the unique properties of the ASH system is that it provides mechanisms for efficient dynamic protocol composition. For example, it can integrate layer processing steps at runtime. To demonstrate the benefits of protocol composition we built a simple web server that serves HTTP requests. The server uses an ASH built out of three layers for the send and receive path. The receive path consists of three layers for: processing the AAL5 trailer, computing the IP and TCP checksums, and performing an application-specific operation that checks whether the data requested is in the server cache. (The send path is also an integrated ASH composed out of three layers.)

We have also built a UDP version of this simple server by just composing the UDP layer instead of the TCP layer with the AAL5 layer and application layer. Unlike the TCP version, the UDP version does not set up and tear down connections. We have downloaded the UDP version into the kernel and run it as an ASH. The TCP version has also been dynamically composed, but has only been tested in user space.

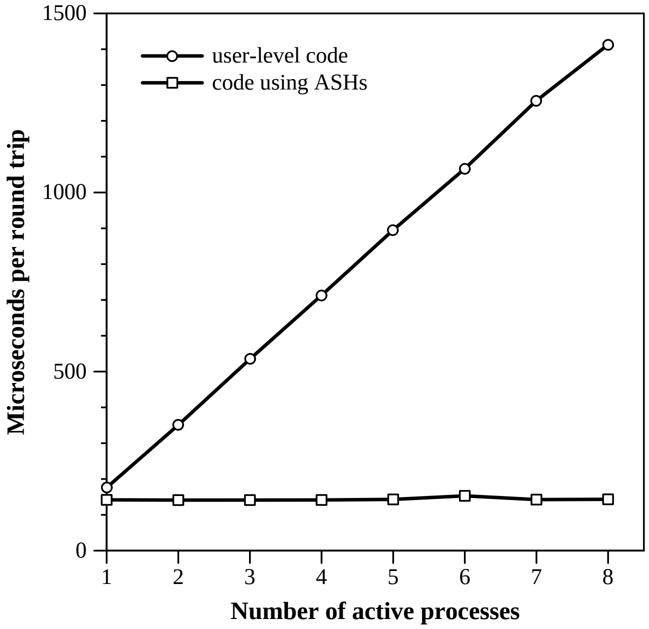

Figure 5: As the number of processes on the system increases, the cost of waiting for a process to be scheduled becomes increasingly higher; times are in microseconds per roundtrip.

The ASH system dynamically composes the three layers, integrates copying and checksumming, and executes the resulting code on message arrival. If the web page is in the main-memory cache, the server sends a reply message directly out of the ASH. If the ASH aborts (*e.g.,* because the requested web page is not in the cache), the message is delivered to the server process, which can load the requested page, install it in the cache, and then reply. Although the server is quite simplistic, its basic operation is like any other server.

We have measured the latency to serve one kilobyte web pages using our dynamically composed UDP server. The latency per request, 660 microseconds, is almost exactly equal to the latency per request of the same server implemented in user space with static protocols. This data shows that even with an untuned implementation of ASHs, protocol composition can be performed at runtime without a performance penalty.

6 Related work

Upcalls ASHs can been viewed as a restricted form of Clark's upcalls [8]. ASHs are intended primarily for simple, small-latency operations; the time they run in can be bounded, because the operating system can reason about their behavior (as well as check for safety). ASHs must be limited in expressiveness to allow the operating system to do this reasoning. Upcalls do not suffer this limitation. Therefore, in cases where a richer set of computations is required, the operating system could perform an upcall to the application at message processing time, instead of calling an ASH (or the ASH could initiate this upcall itself). If the upcall does not complete promptly, the operating system simply interrupts it, saves its state, and continues. While this model is more expressive than ASHs, it has a high computational cost: a full or partial context-switch is required, as well as a number of kernel/user protection crossings. We believe that both of the models can be useful in systems. Those that can tolerate more latency can use the flexibility of the upcall; those that cannot will be confined to ASHs.

Code importation There are a number of clear antecedents to our work: Deutsch's seminal paper [12] and Wahbe et al.'s modern revisitation of safe code importation [44] influenced our ideas strongly, as did Mogul's original packet filter paper [31]. In some sense this work can be viewed as a natural extension of the same philosophical foundation that inspired the packet filter: 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 [21] 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 [22], 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 [37]; in an active network, protocols are replaced by programs, which are safely executed in the operating system on message arrival. Small and Seltzer compare a number of approaches to safely executing untrusted code [35].

Message vectoring Message vectoring has been a popular focus of the networking community [14, 15, 16, 36]. 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.

The most similar work to the ASH system is Edwards et al. [16], 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), limiting the flexibility with which applications can manipulate data transfer. For example, applications cannot synthesize checksumming or encryption functionality. Furthermore, their interface precludes the ability to transfer control or to reply to messages. 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.

In the parallel community the concept of *active messages* [43] has gained great popularity, since it dramatically decreases latency by executing the required 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 [30, 36]. U-Net designed for ATM networks, does provide protection, but only at a cost of higher latency: messages are not executed until the corresponding process happens to be scheduled by the kernel [36]. HPAM is designed for HP workstations connected via an FDDI layer. It makes the optimistic assumption that incoming messages are intended for the currently running process; messages intended for other processes are copied multiple times. The described implementation of HPAM does not provide real protection: they make the assumption that no malicious user will modify the HPAM code or data structures. Our methodology can be viewed as an extension of active messages to a general purpose environment in a way that still guarantees small latencies while also provides strong protection guarantees.

Issues about schedulability and when and how a message handler should abort have been recently explored in Optimistic Active Messages [46]. The tradeoffs discussed there are applicable here.

ILP and protocol composition There have been many instances of ad hoc ILP, for example, in many networking kernels [9]. There is also quite a bit of work on protocol composition [5, 23, 24, 41, 42].

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. Each macro is written with initialization and finalization code and a main body that takes in word-sized input and emits word-sized output. 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 to the system, whereas our system allows untrusted code to participate in ILP in a safe and efficient manner. In one sense this last difference is a practical limitation: static composition makes dynamic extensions to the ILP engine infeasible. Given the richness of possible data manipulations, however, disallowing application-specific operations can carry a significant cost. For example, even a single re-traversal of the data can halve available bandwidth. Proebsting and Watterson describe a new algorithm for static ILP using filter fusion [33].

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. For example, to perform data conversion between two hosts a static system must have pre-composed all possible conversion methods (*i.e.,* between big- and little-endian, external and internal ASCII, Cray floating point and SPARC, etc.). Additionally adding all possible checksum, encryption, and compression operations will only increase code size. 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.

7 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.

Acknowledgments

We thank the DEC Systems Research Center for lending us four AN2 Network Interface cards and a AN2 switch, and for giving us device drivers for these cards. We would also like to thank Mike Burrows, for providing useful and speedy assistance while we were porting the device drivers to Aegis, and Hal Murray, for helping us get the switch running. Thanks are also due to Thomas Pinckney and Héctor M. Briceño for implementing large parts of the support software for interacting with Aegis, and to Eddie Kohler, 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 Transactions on Networking*, 1(5):600–610, October 1993.

[2] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.S. Roselli D.A. Patterson, and R.Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, CO, USA, December 1995.

[3] M.L. Bailey, B. Gopal, M.A. Pagels, L.L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, USA, November 1994.

[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 *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995.

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

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

[7] T. Braun and C. Diot. Protocol implemetation using integrated layer processing. In *ACM SIGCOMM '95*, pages 151–161, Cambridge, MA, August 1995.

[8] D.D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, USA, December 1985.

[9] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[10] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '90)*, Philadelphia, PA, USA, September 1990.

[11] 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*, pages 262–273, Portland, OR, USA, November 1993.

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

[13] P. Druschel, M.B. Abbott, M.A. Pagels, and L.L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[14] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–189, Ashville, NC, USA, December 1993.

[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 '94)*, pages 2–13, London, UK, August 1994.

[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 '94)*, pages 14–24, London, UK, August 1994.

[17] D.R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, USA, May 1996.

[18] 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, USA, August 1996.

[19] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, USA, December 1995.

[20] D.R. Engler, D.A. Wallach, and M.F. Kaashoek. Design and implementation of a modular, flexible, and fast system for dynamic protocol composition. Technical Memorandum TM-552, Massachusetts Institute of Technology Laboratory for Computer Science, May 1996.

[21] M.E. Fiuczynski and B.N. Bershad. An extensible protocol architecture for application-specific networking. In *Proceedings of USENIX*, pages 55–64, San Diego, CA, USA, January 1996.

[22] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 111–118, San Francisco, CA, USA, March 1995.

[23] R. Harper and P. Lee. Advanced languages for systems software: The Fox project in 1994. Technical Report CMU-SC-94-104, Carnegie Mellon University, Pittsburgh, PA 15213, January 1994.

[24] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), January 1991.

[25] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain Resort, CO, USA, December 1995.

[26] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–132, San Francisco, CA, USA, January 1994.

[27] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, 1989.

[28] K. Li. IVY: A shared virtual memory system for parallel computing. In *International Conference on Parallel Computing*, pages 94–101, University Park, PA, USA, August 1988.

[29] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, Ashville, NC, USA, 1993.

[30] R.P. Martin. HPAM: An Active Message layer for a network of HP workstations. In *Proceedings of Hot Interconnects II*, August 1994.

[31] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, USA, November 1987.

[32] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. Technical Report TR96-93, University of Arizona, 1996.

[33] T.A. Proebsting and S.A. Watterson. Filter fusion. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 119–130, St. Petersburg Beach, FL, USA, January 1996.

[34] D.J. Scales, M. Burrows, and C.A. Thekkath. Experience with parallel computing on the AN2 network. In *International Parallel Processing Symposium*, April 1996.

[35] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proceedings of USENIX*, pages 41–54, San Diego, CA, USA, January 1996.

[36] V. Buch T. von Eicken, A. Basu and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, USA, December 1995.

[37] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architecture. In *Proc. Multimedia, Computing, and Networking 96*, January 1996.

[38] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[39] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Separating data and control transfer in distributed operating systems. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, USA, October 1994.

[40] 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 '93)*, pages 64–73, San Francisco, CA, USA, October 1993.

[41] C. Tschudin. Flexible protocol stacks. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '91)*, pages 197–204, Zurich, Switzerland, September 1991.

[42] R. van Renesse, K.P. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of Fourteenth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 138–150, Ottawa, Ontario, Canada, August 1995.

[43] 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 International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Qld., Australia, May 1992.

[44] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Ashville, NC, USA, December 1993.

[45] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Floyd. Implementing real time packet forwarding policies using Streams. In *Proceedings USENIX Winter 1995 Technical Conference*, pages 71–82, New Orleans, LA, USA, January 1995.

[46] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, USA, July 1995.

[47] M. Yuhara, B.N. Bershad, C. Maeda, and J.E.B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, USA, January 1994.