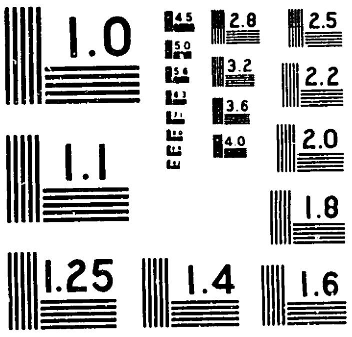

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

High-Performance Application-Specific Networking

by

Deborah Anne Wallach

S.B., Massachusetts Institute of Technology (1990)
S.M., Massachusetts Institute of Technology (1992)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1997

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
31 January 1997

Certified by .... [illegible] ....
M. Frans Kaashoek
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

High-Performance Application-Specific Networking

by

Deborah Anne Wallach

Submitted to the Department of Electrical Engineering and Computer Science on 31 January 1997 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Abstract

Experience with parallel and distributed systems has shown that different application domains require different communication models in order to perform efficiently and be programmed conveniently [37, 57, 60]. Unfortunately, programmers traditionally have had to choose between inflexible but efficient in-kernel protocols and flexible but inefficient user-level protocols. This thesis presents application-specific safe handlers (ASHs), a new technique that provides a protected, efficient, and flexible communication interface.

ASHs make user-level protocols more efficient and usable by enabling application-specific communication protocols to be dynamically yet safely downloaded into the kernel and tightly integrated with the communication system. ASHs eliminate kernel crossings and scheduling delays for common, time-critical events by decoupling network processing from application execution; they also provide the capability for eliminating extraneous data copying and integrating necessary data copying with other data manipulations.

This thesis presents the design of ASHs and their implementation in Aegis, an exokernel-based operating system. It also presents the design and implementation of fast, asynchronous upcalls, an alternative approach to achieving many of the ASH benefits.

Experiments show that ASHs and upcalls can deliver efficient yet flexible communication to applications. Microbenchmarks and end-to-end experiments demonstrate that ASHs outperform upcalls in the cases where the overhead to make them safe is not too high, or where there is a low percentage of successful handler invocations. User-level protocols which use either ASHs or upcalls are shown to outperform user-level protocols which use neither when there is more than one active process simultaneously using the system.

Finally, the thesis presents an analytical model which predicts how the performance would change for different operating systems, architectural models, and application characteristics. We predict that ASHs and upcalls are likely to have even better performance over a wider range of application characteristics in traditional operating systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

My research career was greatly affected by Frans Kaashoek, not only my advisor but a friend as well. He coaxed me out of parallel computation into a whole new area (systems research), a move good both professionally and emotionally. I have very much enjoyed the process of becoming a researcher with his guidance.

I also thank the rest of my committee, Dave Clark and Bill Weihl, for providing encouragement and useful feedback throughout my research, both aiding my dissertation, and allowing me to have a timely defense. John Guttag, although he was not part of my committee, was extraordinarily helpful with my job application process, and always found time for me in his busy schedule.

I thank the DEC Systems Research Center for lending us four AN2 Network Interface cards and an AN2 switch, and for giving us device drivers for these cards. I am especially grateful to Mike Burrows, for providing useful and speedy assistance while I was porting the device drivers to Aegis, and Hal Murray, for helping me get the switch running. Thanks are also due to Dorothy Curtis, of MIT, for keeping the DECstations running.

Dawson Engler admirably put up with the stress of simultaneously being my officemate, research partner, and friend, even if he did think that my workouts were missing a few key exercises. His implementation of Aegis provided a platform on which I could do my research; additionally the Dynamic Integrated Layer Processing compiler was solely written by him. Many of the ASH ideas were developed in collaboration with Dawson, and this dissertation greatly benefited from both discussions and work with him.

I also thank the rest of the PDOS group for providing a fun yet challenging place to work. It was eminently clear that if I could survive a presentation of my work to the group, I was more than set for any other audience. Greg Ganger always had time to answer my questions, and provided helpful advice about improving my research. Although I understood very few of his compiler nightmares, Max Poletto was always ready with a new one to discuss, and has certainly completely redeemed himself in the plant kingdom by now. Eddie Kohler wrote the sandboxer used in this thesis, helped with improving the writing quality of papers written on the content of this thesis, and also was the best source of non-technical discussion around. Dave Maziéres could always be counted on as a source of entertainment. Anthony Joseph answered many many computer-related questions for me, especially, but not exclusively related to crusty DECstation knowledge. Last but not least, I am grateful to Thomas Pinckney and Héctor M. Briceño for implementing large parts of the support software for interacting with Aegis, and for providing for providing assistance with it at all times of the night.

I thank my long distance friends, Carrie Brownhill, Ike Chuang, Brian Murphy, and especially David Kramer, for keeping me sane and giving me an excuse to come in to work. No matter what the season, I could always count on at least one of them to let me know what gorgeous weather I was missing in California.

Finally, I thank Fred Chong, without whose care and support I would have found graduate school a very lonely place.

Some of the text of this thesis was taken from previous papers written by the author and has appeared elsewhere. Specifically, large parts of Chapters 2, 3, 4, and 6 are from [64]. In addition, parts of Chapter 6 appeared in [30, 65].

This research was supported in part by ARPA contract N00014-94-1-0985, by a NSF National Young Investigator Award to Prof. Frans Kaashoek, by an Intel Graduate Fellowship Award, and by Digital Equipment Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Raw processing power and raw networking performance are increasing with time. Although the performance available today may be enough to satisfy the needs of applications written a few years ago, the applications being written today are, commensurately with increases in hardware performance, more complex and demanding in their requirements. For example, the last few years have seen a proliferation of distributed shared memory systems [30, 32, 35], real-time video and voice applications [63], parallel applications [14, 47], and tightly-coupled distributed systems [2, 60, 56]. 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 [13, 16, 56]. This thesis 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 without ASHs. ASHs are also compared in this thesis to fast, asynchronous upcalls, an alternative approach to achieving many of the ASH benefits; microbenchmarks and end-to-end experiments demonstrate that ASHs outperform upcalls in the cases where the overhead to make them safe is not too high, or where there is a low percentage of successful handler invocations.

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). 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 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 can dynamically control 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 (*e.g.*, an application data structure).

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.

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 [61] or remote lock acquisition in a distributed shared memory system. Even recently, low-overhead control transfer had been considered to be infeasible to implement [56].

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 costly memory traversals, stressing a weak link in high-performance networking: the memory subsystems of the endpoint nodes. As argued by Clark and Tennenhouse [13], 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.

In this thesis we also introduce a model for predicting the performance advantages achievable by using ASHs as well as two competing strategies for delivering messages to applications (upcalls and unaugmented user-level communication). The model takes as input a variety of hardware and operating system-dependent parameters, and then can be used to predict which strategy to use for a given application, once given the application-dependent parameters: how long the handler will run for, what the penalty is to make that handler code safe, and how often the handler will succeed.

1.1 Motivation

Applications require flexibility in their communication options. For example, different applications need different communication models. Some applications require reliable, ordered communication: they never want to receive any out of order packets, nor do they wish to miss a packet. On the other hand, other applications, such as ones which present streams of voice data to users, have mostly real-time constraints. It is relatively easy to mask a missing packet (a human will usually not be able to detect a missing packet out of an audio stream), but a delay is a disaster (it is very easy for a human to detect a pause in an audio stream). Another application domain is parallel applications running on top of networks of workstations using a software distributed shared memory (DSM) or other communication models for communication. It has been shown that the performance of such parallel applications greatly depends on the cache coherence protocols used by the DSM implementation. Other applications have even wider requirements for flexibility. Web servers all depend on TCP, a standard connection-oriented, reliable, Internet protocol. A Web server can improve its performance by using an optimized version of TCP, but can achieve an even greater level of performance by using a version of TCP that is actually integrated fully with the server. For example, combining the disk buffer cache with the TCP retransmission pool results in greater system performance, because data does not have to be stored in multiple locations [31].

1.1.1 Kernel-level vs. user-level communication

There are currently two communication options available to applications running in a distributed system environment: in-kernel protocols and user-level protocols. In-kernel protocols, such as a version of TCP built into the operating system, are the traditional way to provide communication protocols, and appear in nearly every operating system today. They may be highly efficient for particular uses (*e.g.*, TCP has been optimized for throughput on many systems), but are inflexible (applications cannot modify or specialize them), and therefore cannot meet the needs of every application. Furthermore, because the set of protocols inside of an operating system is changed only when the operating system is changed, they can be upgraded to meet the needs of new applications only highly infrequently. Again, the web provides us an example of an application with this problem: a web client and server. They communicate using TCP, an extremely common protocol, but use it in a new way: every HTTP request/response pair sets up a new TCP connection (and then closes it when it is done). Because TCP was optimized under the assumption that connections last a long time, the speed of opening and closing connections has not been optimized. In addition, because TCP is located in the operating system on most systems, nearly no one can get a faster version of TCP without upgrading or patching their

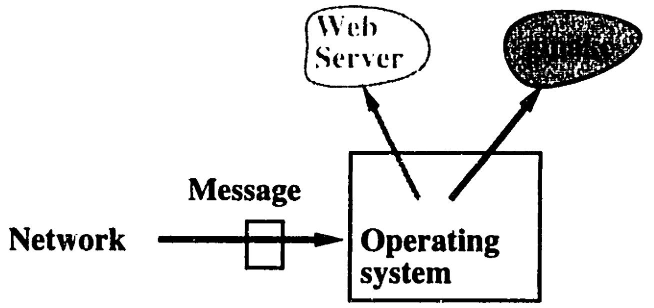


Figure 1-1. In this figure, the gmake application is running when a message for the web server arrives. In order to handle the message, a traditional operating system will (at best) suspend the gmake application and switch in the web server, which is a fairly expensive operation. At worse, the operating system will not even preempt the gmake, but will instead wait for it to finish running.

operating system, an act that many are loathe to do.

User-level communication was proposed in order to solve these problems [41]. The idea is that all communication code is placed outside the operating system, and each application can have its own communication protocols which it can optimize to its own needs. The role of the operating system is just to provide protection between the applications, as opposed to providing heavyweight communication abstractions to the applications.

1.1.2 Problem

User-level communication is an excellent way for applications to achieve flexibility in communication. It does, however, suffer from two problems: the use of user-level communication, as it stands today, limits both the functionality and the performance available to applications. Functionality is impaired because communication code running at the user level is not integrated with the operating system. For example, some software DSM systems are integrated with virtual memory[10, 32, 35]. Because virtual memory is normally implemented in the operating system, it can be awkward and slow for user-level code to query or modify it.

The performance of systems built with user-level communication can suffer in two ways. The first is that applications may see high latency for roundtrip communication if they are not scheduled when a message arrives. An example of this is shown in Figure 1-1. This is a serious problem for user-level communication systems. For example, the roundtrip time for a short message (up to 40 bytes) in U-Net increases from 65 microseconds to 125 microseconds if the applications performing the communication are not scheduled when the message arrives [60]; in Hamlyn the time increases from 28 microseconds to 78 microseconds [9]. Both U-Net and Hamlyn are implemented on operating systems which will suspend the currently running application and immediately reschedule the application an arriving message is for, if the application is suspended waiting for the message. Other operating systems are even less responsive, and will not preempt a running application in response to an incoming message [51, 33].

The performance of systems using user-level communication may also suffer for technology-specific reasons. Certain network interfaces (*e.g.*, many Ethernet cards) deliver messages only to a limited region of memory. It is very important that the messages not stay there too long, or these limited regions will run out of space, and new messages will be dropped. This constraint requires that messages are copied out of the locations that they arrive at, and placed somewhere else. In order to avoid excessive copies, one would ideally like to copy these messages to the final location that the application wants them. However, since the application may not be running when the message arrives, the operating system must perform this copy in order to ensure that the network interface does not run out of room; the operating system, however, does not have application-specific knowledge of where best to place these messages. In contrast, if there were an in-kernel protocol handling the message, then the in-kernel protocol would always be available to immediately direct the message to be copied to a desired location. On the other hand, the in-kernel protocol may not be able to ensure that this location is desirable for the application, so another copy may be necessary.

1.1.3 Solution

We would like to provide the performance of the in-kernel protocols with the flexibility of the user-level protocols. In order to accomplish this goal, the key question to consider is what types of operations should be supported in response to a message. To achieve the maximum flexibility, an application should be able to perform a variety of steps in response to a message. The first is to control where a message is copied, if it does need to be copied. If the message is being copied, the application should be able to specify other operations to perform as well during the copy (for example, as long as it is reading through the message, it should be able to do a checksum or byteswap, etc, on the message data if desired). An application should also be able to perform some computation in response to a message, for example a parallel reduction operation might need to combine some of the message data with a local value in order to produce a new result. Finally, an application should be able to respond to an incoming message with a message of its own; this ability allows the system to provide low-latency roundtrip messaging.

Supporting all of these types of operations provides a powerful programming model for applications to control their communication needs. It is possible to provide this support, albeit at differing amounts of efficiency, using three communication mechanisms: ASHs, fast upcalls, and unaugmented user-level communication, as shown in Figure 1-2. Many of the benefits attributable to ASHs can also be achieved using upcalls, but as will be shown in this thesis, each is appropriate for different situations (depending on the hardware, operating system, and application characteristics).

An ASH, as previously described, is user code (also referred to as a *handler*), downloaded into the kernel and run in response to a message. The version of *upcalls* that we refer to is a handler run in response to a message in user-space (*i.e.*, not downloaded into the kernel). This version of upcalls that we provide is *asynchronous*: the upcall may be to an application that is not running. Implementations of fast synchronous upcalls (*i.e.*, exceptions) are common

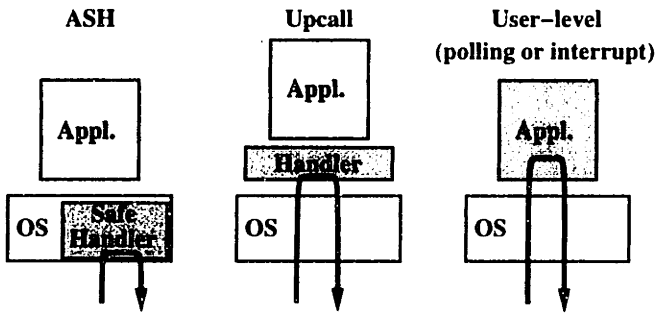


Figure 1-2. This figure shows the three communication mechanisms that we contrast in this thesis. The leftmost one is ASHs, in which application code is made safe and then downloaded into the operating system and run in response to a message. The middle diagram shows upcalls, in which application code is run at user-level in response to a message. The right diagram shows normal user-level communication, where the application may be currently running and polling for a message, in which case it can immediately receive the message, or suspended waiting for a message, in which case it must be rescheduled before it can handle the message.

and easy to add to an operating system; implementations of fast asynchronous ones are neither common nor trivial to implement [36]. Although a fast upcall requires a switch to user space to run the handler, a full process switch is unnecessary; this is what provides us with the speed. In contrast, normal user-level communication involves a full process switch to the application, if it is not running when a message for it arrives.

1.2 Contributions and results

The goal of this thesis is to explore how to provide a timely, efficient response to incoming messages. The mechanism that we propose is a handler, an application-specific piece of code that can be run in response to the message. The contributions of this thesis are the techniques required to support handlers and an evaluation of these techniques:

- **ASHs** We present the design and implementation of ASHs, a technique for delivering hardware-level network performance to applications by downloading application-specific code into the operating system. As described earlier in this chapter, ASHs allow direct, dynamic message vectoring, message initiation, and control initiation.

- **Fast upcalls** We also present the design and implementation of fast, asynchronous upcalls, an alternative approach to implementing handlers. Our design of upcalls is based on the ASH design, and can therefore achieve many of the benefits available to ASHs.

- **Dynamic ILP**[1] This thesis also presents dynamic integrated layer processing (DILP). Through the use of DILP, data manipulations such as checksumming or conversions can be automatically integrated into the data transfer engine itself. DILP can be used with either ASHs or fast upcalls.

- **Sandboxing kernel-level code**[2] This thesis also provides a detailed description of the techniques necessary to run application code in the kernel safely even when the application programmer is not constrained to write in a pointer-safe language. Although the concept of sandboxing is not new [62], we introduce the concept of avoiding exceptions, discuss a variety of techniques for bounding execution time, and limit the available operating system interface in order to restrict the amount of OS changes that need to be made to support application code running in the kernel.

- **Experimental evaluation** The thesis experimentally evaluates three different strategies to provide flexible communication to applications: unaugmented user-level communication, fast upcalls, and ASHs using both microbenchmarks and end-to-end applications. It also demonstrates that ASHs and upcalls can provide many of the benefits of in-kernel communication without the loss of flexibility such a model usually requires. For example, the coherence protocol of a software distributed shared memory system can be changed without changing the kernel itself.

- **Analytical model** Finally, we develop and present an analytical model of these three different strategies which allows us to examine the tradeoffs as the architectural model, operating system, and application characteristics change.

As this thesis shows, both upcalls and ASHs can enable user-level communication to achieve high performance even in situations where it was previously unable to do so, by ensuring that incoming messages can be responded to both efficiently (in terms of system resources) and quickly (so as to maximize application performance). There are tradeoffs between the use of these two techniques, however. ASHs require less overhead to initiate (the application code can start running faster) and are more tightly coupled to the operating system; depending on the architecture, they may incur a fair bit of sandboxing overhead to make safe. Upcalls are slower to initiate (since they must cross a protection boundary), but incur no sandboxing overhead. Additionally, fast upcalls may be more difficult to implement efficiently in a traditional operating system, because they require a domain crossing, and may therefore interact with a larger portion of already-existing code than do ASHs. The most significant result of this thesis, however, is that it shows that avoiding scheduling, whether through the use of ASHs or upcalls, provides a timely response to messages.

[1]This work was done with Dawson Engler.

[2]The implementation of the sandboxer was done by Eddie Kohler.

1.3 Overview of the thesis

The remainder of the thesis is structured as follows. Chapter 2 delves more deeply into the work related to this thesis. Chapter 3 explains the programming model for ASHs. Chapter 4 describes the techniques we use to ensure that ASHs and upcalls are safe and considers what requirements this places on the operating system. Chapter 5 specifies the interfaces used by the different software subsystems described in the thesis and describes Aegis, the operating system we used as our platform. In Chapter 6 we present the experimental results of the thesis, after first describing the full experimental environment and methodology, using both microbenchmarks and full applications. Chapter 7 presents the analytical model we developed, and uses parameters measured from two platforms to explore the tradeoffs among the various communication models discussed in the thesis. Finally, Chapter 8 concludes the thesis, offering suggestions for areas of future work.

Chapter 2

Related Work

The work related to this thesis falls into four classes. The first class is the model of structuring a system to be message-driven; we use handlers to provide this model. Making application code safe is becoming a popular focus in the operating system community; these efforts are related to our work on safely downloading ASHs into the operating system. The benefits provided by executing code in response to a handler (message vectoring, control initiation, and ILP) have been explored, partially, by researchers in other contexts. Finally, the issue of handlers and how they affect the scheduling of the system as a whole is also important to consider.

2.1 Computational model

ASHs can been viewed as a restricted form of Clark's upcalls [11]. Upcalls were proposed by Clark as an alternative way of structuring systems. Instead of having asynchronous communication between layers in a system, where one layer processes the message then later the next one does, etc., he suggested using synchronous communication, or procedure calls, with the lower layers invoking the higher ones directly. ASHs (and fast upcalls as we have proposed them in this thesis) keep the same philosophy of responding to a message by invoking higher layers from the lower ones directly.

There are a number of ways to implement Clark's upcall philosophy. His implementation of them was for a system with a single address space, and used a high level language which required garbage collection, clearly a different design point than we are aiming at. Modern operating systems implement upcalls (or *exceptions*), but only synchronously (*i.e.*, only to a running process). Liedtke's μ-kernel implements extremely fast, asynchronous upcalls by performing address space switches instead of full context switches [36]; it is this type of upcall that we compare ASHs to in this thesis.

Because ASHs are intended primarily for simple, small-latency operations, the time they run in can be bounded, since 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 effectively. 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). While this model is more expressive than ASHs, it has a higher computational cost: an address space 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.

2.2 Safe code importation

There are a number of clear antecedents to our work on making ASHs safe: Deutsch's seminal paper [15] and Wahbe et al.'s modern revisitation of safe code importation [62] influenced our ideas strongly, as did Mogul's original packet filter paper [41]. 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 [5] is concurrently investigating the use of downloading code into the kernel. SPIN's Plexus network system runs user code fragments in the interrupt handler [25] 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 [26], 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 [53]; 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 [50].

The Vino project is also investigating means for safely importing code into the kernel [48]. They consider five classes of misbehaving application kernel extensions, and introduce a transactional model to prevent these types of misbehavior. Our framework also prevents these classes of misbehavior, but instead of using a heavyweight transactional model, we carefully restrict the interface. 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.

Finally, Necula and Lee are investigating Proof Carrying Code, a method where code carries with it a proof that it is safe [43]. The operating system can then check the proof at download time, and therefore there is no need for runtime checks (except as an aid to generating a proof). So far they have only been able to automate proof generation for very small examples, and have included programmer-written loop invariants for each loop (to make the proof generator simpler), but we regard their work as a very promising direction to making ASHs run faster (*i.e.*, by reducing the sandboxing overhead).

2.3 ASH benefits

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

2.3.1 Message vectoring

Message vectoring has been a popular focus of the networking community [17, 18, 20, 60, 46]. 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) [18] 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. If the receive queue of an ADC is empty when a new message arrives, the interrupt handler will signal a thread of the driver. 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. [20], 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.

2.3.2 Control initiation

In the parallel community the concept of *active messages* [61] 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 [38, 60]. U-Net, originally 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 [60]. 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 providing strong protection guarantees.

2.3.3 ILP and protocol composition

There have been many instances of ad hoc ILP, for example, in many networking kernels [12]. There is also quite a bit of work on protocol composition [6, 27, 28, 58, 59].

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 [45].

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.

2.4 Scheduling

An improperly designed operating system can suffer from receive livelock when faced with a constant stream of network interrupts, as described in Mogul and Ramakrishnan [40]. Correctly adding ASHs to an operating system which has no receive livelock will not reintroduce the problem. To avoid livelock, the operating system must keep track of how many ASHs have been recently executed on a per process basis, and refuse to execute any more for processes receiving more than their share of messages, instead falling back to the normal mechanism.

A similar approach to fairly and stably dealing with high communication loads is described in Druschel and Banga [19]. The two key techniques described in their paper are 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. 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.

Issues about schedulability and when and how a message handler should abort have been recently explored in Optimistic Active Messages [65]. The tradeoffs discussed there are applicable here.

Patrick Sobalvarro has explored *demand-based coscheduling*, where processes that are communicating with one another on different machines are scheduled simultaneously [51]. This strategy greatly increases the likelihood that an application will be running when a message arrives for it if the application is communicating often with the other applications it is interested in being coscheduled with. This idea should be very useful for particular types of applications, such as parallel applications, but not for ones with less predictable communication patterns (such as a web server). Additionally, ASHs are still beneficial in some cases as compared to applications already running at message arrival.

Chapter 3

ASH Programming Model

ASHs and upcalls both use the model of a *handler*, a fragment of application code specially designated to be run in response to an incoming message. This chapter discusses how the system determines which handlers to run, the different types of handlers, and what it means when a handler cannot handle a message: in other words, the programmer-visible interface available for handlers. This chapter discusses these issues in terms of ASHs; the same interface can be provided for fast upcalls, however. The next chapter considers an ASH-specific problem: how to take an ASH and make it *safe* so that it may not harm other applications or the operating system.

3.1 Message demultiplexing

In any user-level communication system, there must be some way for the operating system to determine which application a message is for. There are a variety of standard techniques available to perform this demultiplexing. One such technique is *packet filters* [41], in which applications specify patterns describing the packets that they are interested in (*e.g.*, all IP packets sent to port 80) and an in-kernel packet filter engine examines each incoming message, determining the matching application and delivering the message to it. Packet filters can be implemented efficiently [22].

Another technique sometimes used by user-level communication systems implemented on top of ATM networks is to use the virtual circuit index of the incoming message as an application identifier [60].

Any technique used by user-level communication to demultiplex messages may also be used by user-level communication augmented with ASHs or upcalls. In addition to specifying the demultiplexor, the application must also specify the handler code to be associated with the demultiplexor; then when a message arrives that matches the demultiplexor, the associated handler can be run.

3.2 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.

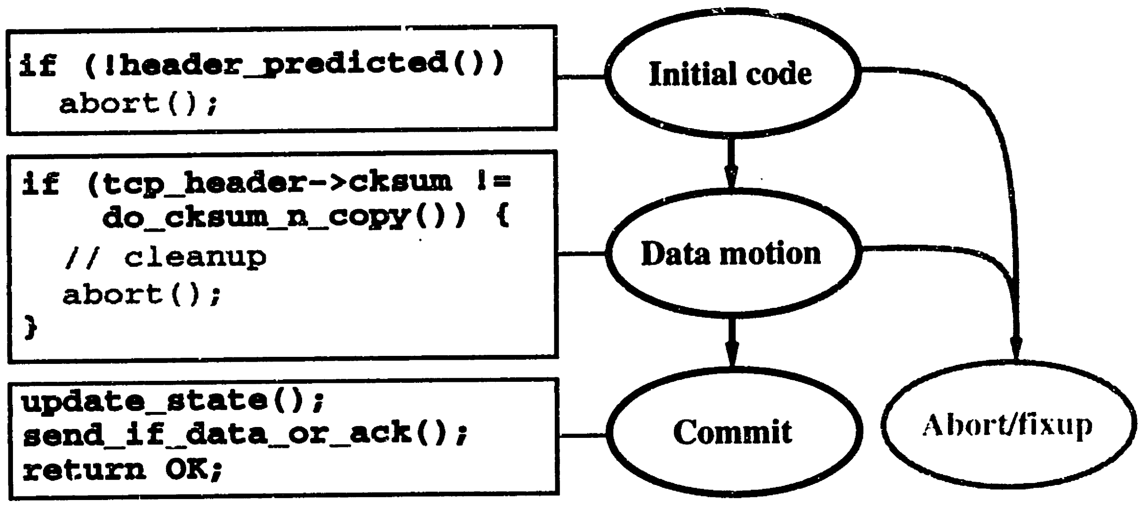
ASHs are user-written routines that are downloaded into the kernel to efficiently handle messages. From the kernel's point of view, an ASH is simply code, invoked upon message arrival, that either consumes the message it is given or returns it to the kernel to be handled normally. From a programmer's 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 thesis 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.

3.2.1 Static ASHs

All types of ASHs are generally written in a stylized form consisting of three parts, as shown in Figure 3-1. 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



```
if (!header_predicted())
  abort();
```

```
if (tcp_header->cksum !=
    do_cksum_n_copy()) {
  // cleanup
  abort();
}
```

```
update_state();
send_if_data_or_ack();
return OK;
```

Figure 3-1. This figure shows the stylized form that all types of ASHs are generally written in, and uses an ASH built for the TCP protocol as an example to illustrate the form. The initial part of the ASH tests whether or not the arriving message is one that TCP expects (*i.e.,* whether or not its header is the one that TCP predicted). If it is not, the ASH will not handle it and will instead choose to abort and pass the message up to the main application code. If it is, the ASH will continue to the data manipulation portion. In the case of TCP, the ASH performs a checksum and a copy of the data to application data structures. If the checksum fails, then the message is useless; the ASH will choose to abort and will discard the message. Assuming the first two stages complete successfully, the ASH can then modify any state that it needs to and send out any appropriate responses, if any.

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.

3.2.2 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 [13] 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 possible operations, such mismatches happen quite easily. Furthermore, many systems compose protocols at runtime [6, 27, 28, 58, 59], 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.

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 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

```
// Initialize a pipe list for two pipes (checksum and byteswap)
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 3-2. This code fragment composes to pipes into a pipe list, and then compiles that list. The pipe creation functions take a pipe list as input; the checksum pipe creator function also requires as input an accumulation register it can use to calculate the checksum into (`pipe_cksum` in this case). The compile function takes a pipe list and a flag indication whether or not the pipe list will require different source and destination locations for the message.

16-bit checksum pipe's output to be streamed through a 32-bit encryption pipe, it is aggregated into a single register.

Figure 3-2 presents an example composition of the checksum pipe of Figure 3-3 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 [21], 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 [7] is provided in Figure 3-3. 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

```
// Specify a pipe to compute the Internet checksum and return its identifier.
// This specification is subsequently converted into safe machine code.
// 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);
        // Allocate an accumulate register (preserved across
        // pipe applications)
        reg = p_getreg(pl, pipe_id, P_VAR);
        // Get 32 bits of input from the pipe
        p_input32(p_inputr);
        // Add input value to checksum accumulator
        p_cksum32(reg, p_inputr);
        // Pass 32 bits of output to next pipe
        p_output32(p_inputr);
    pipe_end();
    *cksum_reg = reg;
    return pipe_id;
}
```

Figure 3-3. This function generates the specification of a pipe to be used in the computation of the Internet checksum, and returns the identifier of the specification. `pipe_lambda` indicates the start of the pipe description, and `pipe_end` the end of it. The pipe will need to be added to a pipe list and compiled before it can be used. When invoked, the pipe will consume a 32-bit word of data, add the value of that data to the running checksum total along with any overflow, and then output the unchanged input.

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 Figure 3-3). 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.

Figure 3-3 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-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 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 bits.

3.2.3 ASHs with dynamic protocol composition

In addition to dynamic ILP, ASH programmers can use the dynamic protocol composition extensions provided by the ASH system [24]. 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 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; 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 3-4 shows the main body for a very simple ASH that implements naive remote writes [56]. 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.

An example of a system which could employ dynamic protocol composition is a server, which would use integrated ASHs for both 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.

This thesis addresses only the design of ASHs with dynamic protocol composition, and not the implementation. Although an initial prototype has been completed, it is too preliminary to evaluate the feasibility of dynamic protocol composition. Providing programmers with a means to write modular yet efficient code is an important problem, but is orthogonal to the main issues of the thesis.

```
// 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;
    // Load destination address (first word)
    if (!consume(&dst, sizeof(char *), NULL_PIPE))
        // Short message
        return ASH_ABORT;
    // Load length (second word)
    if (!consume(&len, sizeof(int), NULL_PIPE))
        // Short message
        return ASH_ABORT;
    // Incomplete packet
    if (nbytes < len+sizeof(int)+sizeof(char *))
        return ASH_ABORT;
    // Copy len bytes from (message buffer + 8) to (dst).
    if (!consume(dst, len, NULL_PIPE))
        // Short or corrupt packet
        return ASH_ABORT;
    // Success.
    return ASH_SUCCESS;
}
```

Figure 3-4. Example ASH using the dynamic protocol composition extensions.

3.3 Aborts: terminating ASHs

ASHs were designed to handle only the common case. They must therefore be able to terminate specially in the uncommon case, when they cannot handle the message. Other than a normal return, ASHs can terminate in two ways: *voluntarily*, at their request, or *involuntarily*, at the system's demand.

3.3.1 Voluntary aborts

Voluntary aborts were described earlier in this chapter. If an ASH decides it cannot handle a message (for example because handling the message would cause it to run too long), it can choose to abort. The ASH itself is responsible for undoing any changes it has made in order to keep the application state consistent. We chose this particular model in order to keep the overhead of running ASHs small (*i.e.,* this way the system does not need to keep track of every action an ASH makes). All types of ASHs can choose to abort voluntarily. If an ASH does not indicate to the system that it has handled the message at the time of termination, the message will be delivered to the user process at the next opportunity.

3.3.2 Involuntary aborts

An ASH is obviously only allowed to perform certain actions, because it is code running in an operating system. If it should attempt actions disallowed by the ASH interface, however, and these actions could not be detected statically, the system must take some action. Because these actions indicate it to be incorrect code, the system has a choice of options it may employ. Certain incorrect behaviors can be easily prevented without detection, and in fact detecting them is more expensive that preventing them, as will be discussed further in Section 4.1.2. We therefore choose to allow an ASH which attempts such an action to continue running, but prevent the incorrect action. On the other hand, some incorrect behaviors are just as simple to detect as to prevent. If an ASH attempts one of these behaviors, we reserve the right to terminate it immediately at that point, using an involuntary abort. Again, because this represents the behavior of an incorrect ASH, we are not worried about the subsequent execution of the application. A programmer who writes correct code will never have an ASH terminate involuntarily.

3.4 Summary

The preceding sections described the design of a system to allow application programs to handle messages rapidly and efficiently. The design is based around the concept of a handler: a fragment of application code specially designated to be run in response to an incoming message. There are a variety of different programming interfaces to handlers that can be offered to programmers; three were discussed in this chapter. The next chapter describes how to take these application-specific handlers and make them safe, so that they cannot harm the operating system, or other applications.

Chapter 4

Safe Execution

There are a number of interesting issues associated with running application code immediately in response to a message, either through the ASH or upcall mechanism. The first is to provide safety: how to ensure that the handler does not harm other applications or the operating system. A related issue is how to ensure that the handler does not run too long. In this chapter, we discuss these two issues first for ASHs and then for upcalls.

This chapter presents a variety of software and hardware techniques that go beyond the ones originally proposed by Deutsch and Grant [15] and Wahbe et al. [62]. Because they do not specifically address the issue of downloading code into the operating system, they have not explored the issues of avoiding ASH exceptions and bounding ASH and upcall execution time; similarly, they do not discuss constraining the operating system interface.

4.1 ASHs

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: first the requirements we place on the operating system and then our strategies for executing ASHs safely.

4.1.1 The operating system model

The most important task required 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.

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 non-resident page, or an illegal memory address, it is aborted.

For efficiency reasons, we could 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.

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 4.1.2, 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.

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

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

Exceptions are prevented using runtime and static checks (as is done in existing packet-filters [41, 67]). Wild memory references and jumps are prevented using a combination of address-space fire-walls and *sandboxing* [62]. The ASH system for the MIPS can bound execution time using a framework inspired by Deutsch [15]. We examine each technique in further detail below.

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 in no code that we have sandboxed has there been any signed arithmetic instructions generated by the C compiler[3]). 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 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; 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 inspection (*sandboxing*) techniques of Wahbe et al. [62].

Making sandboxed data copies efficient requires complex analysis of the user-supplied code. The ASH system therefore provides 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

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

[3]This means we are precluding the use of Fortran for writing ASHs, since Fortran compilers generally use signed arithmetic instructions, but we do not consider this to be a great loss

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.

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 thesis 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 which contain loops, software checks at all backwards 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 which 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).

4.2 Upcalls

Upcalls require less system support than ASHs do. Like ASHs, upcalls need address translation. This translation is supported identically to that of ASHs for our MIPS implementation: the context identifier and pointer to the page table of its associated application must be installed. Upcalls are not aborted at references to non-resident pages; illegal page references cause the same operating system-generated signal that a normal user-level page reference would. We cannot provide the ability for upcalls to access physical addresses (unlike ASHs, where we could but we have chosen not to), because code running outside of kernel mode on the MIPS that references physical addresses causes an exception.

Because upcalls execute code in user space, most forms of malicious and buggy operations can be detected by normal operating system and hardware mechanisms. The main safety issue for upcalls is thus ensuring they do not execute for too long of a period of time. Since the handler run by an upcall is not pre-approved, we cannot insert checks into the code to ensure that there are no loops, nor can we place software checks at backwards jumps. Instead, we must use a hardware timer in order to seize control back from an upcall that runs too long.

4.3 Summary

This chapter addressed the issue of how to take application code and safely run it in response to a message. We presented both the general design and the MIPS-specific implementation of a system to systematically ensure that both ASHs and upcalls are safe, in the sense that they cannot harm other applications or the operating system. The next chapter presents the implementation specifics of the entire system: networking, the operating system, and the mechanism to provide fast message handling.

Chapter 5

Implementation Specifics

We have implemented a system for ASHs in Aegis, an exokernel operating system [23]. This chapter describes the implementation and testbed environment used in our experiments.

5.1 Overview

As shown in Figure 5-1, Aegis is the operating system we are using as our platform. Aegis provides a low-level interface to the network; it demultiplexes packets to applications, but all other communication code is the responsibility of the application, not the operating system. Figure 5-1 shows applications receiving messages after demultiplexing in three ways. The Web server is using unaugmented user-level communication to receive the message, and will therefore receive the message only when it is scheduled. Barnes-Hut will receive incoming messages as upcalls, unless the upcall handler chooses to pass them up to the application. Finally, FTP will receive incoming messages as ASHs, running in the kernel. The rest of this chapter discusses the various subsystems necessary to support these three communication models.

5.2 Aegis

Aegis is an exokernel operating system for MIPS-based DECstations. The idea of an exokernel [23] operating system is to allow applications to manage the physical resources of a machine as much as is possible. The abstractions and policies normally expected in an operating system (*e.g.*, virtual memory or interprocess communication) are consigned to user-level library operating systems, which execute outside the kernel. In this way, the kernel is only responsible for *protecting* resources, not managing them, allowing applications to specialize, customize, or extend policies and abstractions however they choose.

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.*,

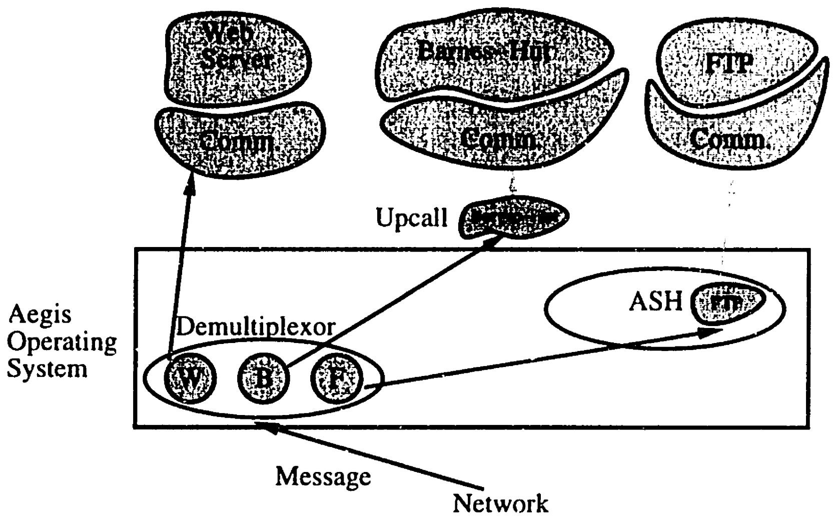


Figure 5-1. This figure illustrates the different subsystems necessary to support ASHs, upcalls, and user-level communication.

TCP/Vegas), server (*e.g.,* Mach network server), or user-level (*e.g.,* U-Net) implementations of networking, although we expect to see the most performance gains on older, more traditional operating systems architectures.

In fact, because Aegis was designed for flexibility and performance from its inception, it is not the ideal platform. As reported in [23], it has extremely fast kernel crossing times (about a factor of 10 faster than Ultrix), and context switches are quite inexpensive (about a factor of 5 faster than Ultrix). As we will see in Chapter 7, this fast performance means that an exokernel operating system benefits less from ASHs than we would expect a traditional one to. Furthermore, because many of the operating system abstractions one would normally expect to be in-kernel are in user space under Aegis, operations that would normally be system calls for user-level communication and function calls for ASHs instead have to be sandboxed and included into the ASH itself (since they are now user-level code).

5.3 Network and operating system interfaces

Aegis provides protected access to two network devices: a 10 Mbits/s Ethernet and a 155 Mbits/s AN2 (Digital's ATM network).

5.3.1 Ethernet interface

The Ethernet device is securely exported by a packet filter engine [41]. The Aegis implementation of the packet filter engine, DPF [22], 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

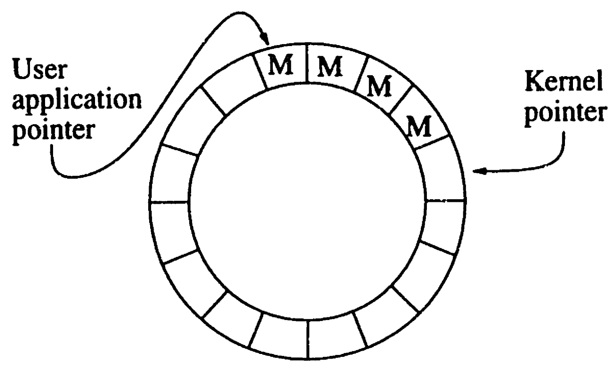


Figure 5-2. The kernel shares a virtualized notification ring for incoming messages with each application. In this figure, the user process is behind in processing messages, so its pointer points further back in the ring than the kernel one does (assuming a clockwise ordering). The kernel's pointer points to the next available slot to place a message notification in. When the user process is all caught up, it can poll on the next empty location in the ring in order to determine if a message has arrived.

filter constants to aggressively optimize this executable code. DPF is an order of magnitude faster than the highest performance packet filter engines (MPF [67] and PATHFINDER [4]) in the literature.

5.3.2 AN2 interface

Similarly to other systems [18, 47, 60], the AN2 device is securely exported by using the ATM connection identifier to demultiplex packets. Before communicating, processes bind to a virtual circuit identifier (VCI). Two kinds of connections are available on our system: receiving from a single host and receiving from multiple hosts. As the many-to-one option is not available on ATM, it is synthesized by the device driver and OS for applications (*i.e.*, input from multiple VCIs is combined into a single notification ring).

As shown in Figure 5-2, the kernel and user share a virtualized notification ring in memory per virtual circuit. By examining the ring, an application can determine that a message arrived and where the message was placed. Applications may poll the next empty location in the ring or depend on an interrupt to receive notification of an incoming message. Unfortunately, there is no mechanism to provide asynchronous notification to running processes currently existing in Aegis; instead, the "interrupt queue" is checked every time a process is re-scheduled (every 15625 microseconds).

At connection setup time, the application chooses the size of this virtualized notification ring (allocated out of its own memory) and notifies the kernel of its size and location. Applications are also responsible for supplying a section of their memory for messages to be DMA'ed to. This memory is fragmented into buffers for the use of the device; when a message arrives it is placed into the next set of buffers for that VCI which has been registered on the NI. The device driver then selects a new set of buffers from the application-supplied pool to replace those buffers with (there are 15 such buffer sets simultaneously available to receive messages in). The application is responsible for replenishing the message buffer stack with buffers, either by consuming the message and returning those buffers to the kernel, or supplying new ones. We have chosen a buffer size of 1024 bytes as a compromise between being able to receive large messages and not to waste buffer space for tiny ones. The vendor-supplied parameters set in the device driver (which we use as well) limit messages to five times the size of this buffer, which for our implementation is 5120 bytes (actually, 5088 bytes, because all messages must be a multiple of 48 bytes, the ATM cell size). These buffers are guaranteed never to be swapped out in our current implementation.

In keeping with the exokernel philosophy, Aegis exports only a low-level send primitive (system call); it is the responsibility of the application to format the message for proper sending. The application provides the VCI to send to, a structure containing pointers to the message fragments to be sent, a message length, and an optional notification flag. The device driver checks that the message format requirements are met and then sets up the transmit; it never copies or buffers the message. The notification flag indicates that the message has been DMA'ed, and that the application can therefore reuse those transmit buffers. The send primitive will return an error if any of the tests for message conformability fail or if the device is too full of packets to transmit (in which case the application is responsible for resubmitting the message later).

There are several requirements placed on messages to be sent, many of which stem from the use of the DMA engine by the AN2 device. The message size must be padded to be a multiple of 48, with eight bytes free for trailer information. All of the message fragments must be four-byte aligned (this is a requirement of the DMA engine). The message may only use a predefined number of physical memory fragments (this number is set to seven in our implementation). If a fragment supplied by the application spans a physical page boundary, the driver will break it into multiple fragments for the DMA engine. The application is also responsible for setting up the AAL5 trailer, excluding the checksum. The device automatically generates and tests the checksum.

In a general-purpose environment, applications would not each be allocated a VCI, as they are in our setup, because there are only a small number of VCIs to be shared between all of the applications. Instead, as in other implementations (*e.g.*, the AN2 device under Ultrix), most would share a single VCI, and a different level of demultiplexing would be used to distinguish between them. For a user-level communication system, the appropriate demultiplexing tool would be a packet filter.

5.4 ASHs

We first describe the ASH interface, then the exact implementation of ASHs for the AN2 interface on Aegis.

System call	Description
ae_gettick	read current clock tick
ae_otto_send	send a message
ae_otto_bump_n_free	return message receive buffers to AN2
ae_pid	get current process id
ae_trusted_memcpy	perform a simple memory copy with bounds check
ae_rapply	invoke a read-only dynamic ILP operation
ae_wapply	invoke a read/write dynamic ILP operation

Table 5-1. System call interface available to ASHs.

5.4.1 ASH interface

In the current system, only static ASHs and ASHs with dynamic ILP are supported. A programmer writes an ASH in C code. The ASH will be invoked as a function which takes four arguments: a pointer to the receive structure that describes the incoming message, the number of bytes in the message, the virtual circuit that the message arrived on, and the slot within the receive ring that the message occupies. The ASH is expected to return an integer value. If the value returned equals one, the system assumes that the ASH has handled the message. Otherwise, the message will be passed up to the user level as it would be if there had been no ASH.

An ASH may only call three types of functions: dynamic ILP invocations (as will be described in Section 5.7), approved system calls, and any C function that also satisfies these criteria. The list of approved system calls is in a system header file, and includes the calls needed to send and receive messages. The full list is in Table 5-1.

The programmer is responsible for locking any data structures necessary so that the ASH and application do not conflict. An ASH obviously cannot wait on a lock, but must instead abort if a lock is not available; this is the responsibility of the programmer to ensure.

5.4.2 ASH implementation

The current system only supports ASHs on the AN2 network interface. When a packet arrives, the AN2 device driver tests whether or not that VCI has an ASH registered for it. If it does not, it proceeds as normal. Otherwise, it invokes ASH wrapper code. The wrapper code switches the system interrupt handlers (so that clock interrupts can be handled specially), and tests whether or not the incoming message is associated with the currently running application. If it is, then the correct context identifier for the application is already set; if it is not, the wrapper code switches the context identifier to that of the correct application. Either way, the wrapper then swaps the TLB fault handlers to one that will ensure that TLB faults will cause the ASH to be aborted, enables clock interrupts (so the ASH will not run too long; interrupts are disabled the rest of the time in Aegis), switches the current stack to an application-specified stack, and invokes the ASH with the arguments described previously.

When the ASH returns to the ASH wrapper, the wrapper undoes the changes it made to run the ASH (*e.g.*, restoring the kernel stack, disabling clock interrupts, restoring the TLB fault handlers and interrupt handlers, and switching the context identifier back). If the ASH returned normally, the device driver checks how many message buffers are still being used and replaces them.

If no buffers were used (because the ASH consumed the message, or moved it to somewhere else), the driver will let the NI reuse those buffers later rather than replacing them; this saves multiple expensive writes across the TURBOchannel bus. If the ASH aborted voluntarily, the system will not reuse those buffers at all, and it will deliver the message intact to the user-level application just as if the ASH had not run.

The ASH may be involuntarily aborted a few ways. If it runs too long (if the system sees two clock interrupts, indicating that the ASH ran for at least a full time slice), the clock interrupt handler jumps to the ASH cleanup code, generated by the sandboxer, which cleans up any callee-saved registers the ASH has used, and returns to the ASH wrapper code. Note that interrupts are disabled during system calls that the ASH makes; this strategy ensures that system calls will not be partially executed. If an ASH takes a TLB fault, the TLB fault handler jumps to the abovementioned ASH cleanup code. Finally, if it attempts to jump to an illegal PC, the jump is changed to a jump to the ASH cleanup code.

5.5 Sandboxer interface

In order to submit an ASH to the system, the programmer invokes the sandboxer. The sandboxer takes as input a pointer to the ASH function, an optional list of support functions to sandbox that the ASH does not refer to directly (the *supports*), and an optional list of functions the invocation of which indicates that an ASH should terminate (the *aborts*). The supports are used to indicate functions which the ASH calls indirectly; since there is no reference to them in the ASH they would not end up otherwise sandboxed, and the ASH would not be able to call them. The aborts are for programmer convenience. A typical way they are used is for assertions. If an assertion fails when the ASH is somewhere in its call stack, the ASH can clean up and gracefully terminate without having to signal an exception all the way back to the top level of the handler.

The sandboxer reads in the ASH function, and sandboxes it. It returns an error to the application if it can statically detect a problem (such as a disallowed instruction, or unapproved system call). If the function is sandboxed successfully, the sandboxer downloads the ASH into the operating system, indicating which portion of it is the cleanup code to invoke if the OS terminates the ASH involuntarily. Currently each process only uses a single virtual circuit identifier, so the ASH is automatically associated with that VCI at download time. This would be trivial to fix.

To ensure safety, the sandboxer should either be running in the kernel, or else running as a trusted process and signing the sandboxed code with a digital signature. Neither case is true in our system, but again this would be simple to fix.

5.6 Upcalls

This section describes both the upcall interface and the implementation of upcalls that was written for the AN2 interface on Aegis.

5.6.1 Upcall interface

The upcall interface is modeled after that of the synchronous interrupt interface in Aegis. Upcalls must be enabled by the application, at which time the application registers a handler to be called. The handler is invoked at upcall time with the virtual circuit of the messages and the slot in the receive ring that the first message of the message batch is located in. The upcall handler may then pull arrived messages out of the receive ring exactly as the normal user-level polling code does. This arrangement allows the system to batch messages, minimizing the number of kernel crossings for a batch of near-simultaneously-arrived messages to a single process.

Just as with ASHs, the programmer is responsible for locking any data structures necessary so that the upcall and application do not conflict. An upcall also cannot wait on a lock, but must instead abort if a lock is not available; this is the responsibility of the programmer to ensure.

5.6.2 Upcall implementation

The current system only supports upcalls on the AN2 network interface. When a packet arrives, the device driver checks if there is an upcall registered on that VCI (ASHs are tested first, for those applications which use both). If not, it proceeds as normal. If so, it enqueues the upcall on a list. When the device driver is done executing (*i.e.,* the hardware interrupt has been handled), the upcalls queued up are executed. Before running any upcalls, the system swaps the system interrupt handlers (again, so that clock interrupt can be handled specially), and sets up the status register so that the user code can execute properly. It then goes through the list of queued upcalls, context switching to a new process if necessary, then jumping to the interrupt handling code (which runs at user level, and will invoke all of the upcalls queued for that process) and executing a Return From Exception instruction (which changes the processor from kernel mode to user mode).

When the upcall returns (using a system call), the next upcall is invoked. If no upcalls are left on the list, the status register is restored to its previous status, the interrupt handlers are reset to their old values, and the correct process is context switched to, if necessary (*i.e.,* the one that was running when the interrupt arrived). At this point, the code returns from the interrupt back to where the processor was executing when the original message interrupt occurs.

If an upcall runs too long (*i.e.,* two clock ticks), it is terminated. There is no provision to clean up at this point; doing so would entail restoring callee-save registers at the minimum. As these registers are not currently saved before invoking upcalls (nor are the floating point ones), the time for upcall invocation should be slightly more expensive than is reflected in our results. In the ASH case, the system depends on the sandboxer to save and restore those callee-saved registers that are used; in the upcall case, there is no way to determine which registers the user code might or might not have touched, and therefore all should be saved before initiating the user code.

5.7 Dynamic ILP interface

As discussed in Section 3.2.2, an application specifies the data manipulation steps it wants integrated in VCODE. It presents this list to the DILP system, which compiles the list down into machine code representing the requested list of data manipulations, and returns an identifier (`ilp` in Figure 3-2). When an ASH or upcall 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 and upcalls 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.

5.8 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 actually performed this analysis. In fact, Liedtke, a researcher who has implemented upcalls highly efficiently, believes that the only way to do so is to completely rewrite the operating system from scratch [36].

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 integrated layer processing, 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[22]). Given that VCODE is a stand-alone package, we find providing DILP to be worthwhile, as DILP greatly simplifies writing efficient integrated loops.

5.9 Summary

This chapter described the specific subsystems that we used to provide efficient user-level communication, ASHs, and upcalls. The next chapter experimentally evaluates the performance of these communication strategies for a series of microbenchmarks and end-to-end applications.

Chapter 6

Experiments

This chapter addresses a main hypothesis of this thesis: that the performance achievable using user-level networking is good, but that handlers improve this performance. We perform a number of experiments to address this hypothesis and others. In addition, this chapter demonstrates several different ways that applications can exploit the flexibility provided by user-level communication.

We first examine the base performance of our system, and compare it to other instances of user-level communication, on other platforms, to show that our implementation of user-level networking is efficient. With this in mind, we then investigate in detail the additional benefits obtainable by using ASHs and upcalls. We show that the addition of ASHs to user-level networking can provide better performance than user-level networking alone, even when there is a only single active process per processor, because ASHs enable high throughput, low-latency data transfer, and low-latency control transfer. We also show that when there are a number of active processes, both ASHs and upcalls can provide better performance than is achievable without them.

We then present a series of applications from three different domains: a basic internetworking protocol, three parallel applications running on top of software distributed shared memory, and a client-server application. These applications together serve a number of goals. First, they demonstrate that our system can support a set of real applications. Second, our TCP implementation demonstrates that dynamic ILP can be effectively used in an application, both to simplify the work of the programmer and to achieve better performance. The parallel applications use a communication model where they expect asynchronous notification of message arrival; both ASHs and upcalls can be used to provide this notification ability for operating systems (such as the exokernel) which otherwise lack it. Our final application is a Web server. This application vividly demonstrates what happens to system performance when there is a large number of aborts.

6.1 Experimental environment

This section reports on the base performance of our system (*i.e.*, without the use of ASHs or upcalls). Like other systems [18, 20, 37, 60, 57], 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.

6.1.1 Testbed

The measurements in this thesis are taken on a pair of 40-MHz DECstation 5000/240s, which are rated at 42.9 MIPS and 27.3 SPECint92, and on a pair of 25-MHz DECstation 5000/125s, which are rated at $\sim$ 25 MIPS and 16.1 SPECint92. All measurements reported in this thesis use the 240s; any measurements including more than two processors use one or two of the 125s, as appropriate. 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. On the 125, the I/O devices are accessed over a 12.5-MHz TURBOchannel bus. The four DECstations are connected with an AN2 switch [3].

6.1.2 Methodology

While collecting the numbers reported in this thesis, we had a fair number of problems with cache conflicts (similar to problems reported by others [42]), 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 thesis 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 many times (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 microseconds, and they are not reported here.

6.1.3 Interrupts

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

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

Table 6-1. Raw latency (in microseconds per round trip) for user-level and in-kernel applications on AN2 and Ethernet.

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.

6.1.4 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 6-1 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 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 microseconds [47], 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 70 microseconds, which brings the total software overhead to 86 microseconds, or about 3,440 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.

Figure 6-1 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) [47]. 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. [47] 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 (182 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 [60]. Direct comparisons with other high-performance systems such as Osiris [18] and Afterburner [20] are difficult since they run on

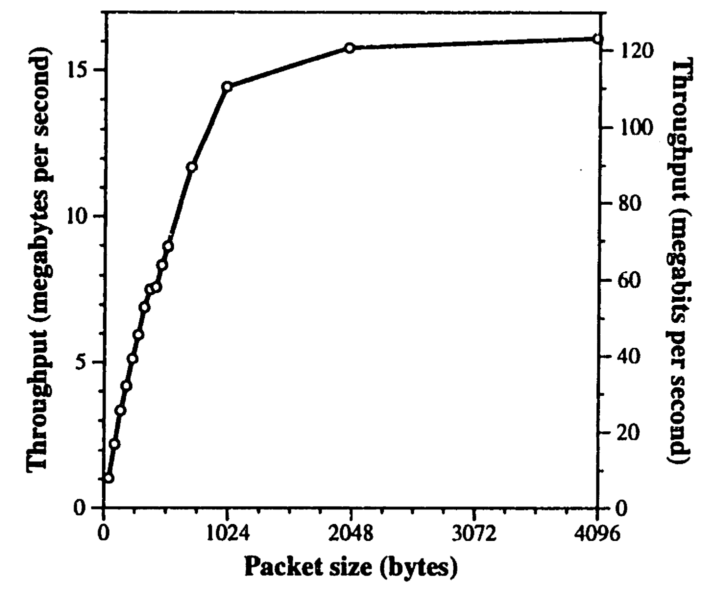


Figure 6-1. Throughput for a user-level application on the AN2.

different networks and have special purpose network cards, but our implementation appears to be competitive.

6.1.5 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 [20, 60]. 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 6-2 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

Implementation	UDP		TCP	
	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

Table 6-2. 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 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 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 do 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 a hard-coded in-kernel implementation.

The leftmost columns of Table 6-2 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 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 processor speed differences) about the same as the fastest implementation reported in the literature [54]. Using the AN2 interface, UDP latencies are about 43 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 [60]; the U-Net implementation adds 73 microseconds on a 66-MHz processor while our implementation adds 62 microseconds on a 40-MHz processor (even though, unlike their numbers, our checksum and memory copy are not integrated for this measurement). In contrast, UDP running over Ultrix on our platform requires about 1500 microseconds 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

Generic untrusted	Application-specific (ASH)	Application-specific (unsafe ASH)
68	38	10

Table 6-3. Dynamic instruction count (excluding data copying) for three implementations of remote write to be run in the kernel.

the full network bandwidth.

The rightmost columns of Table 6-2 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 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 TCP 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 140 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 about the same or is better than most high-performance user-level and in-kernel implementations [18, 20, 25, 37, 57], as long as the applications are scheduled when the messages arrive.

6.2 Sandboxing overhead

Before we examine any ASH results, it is instructive to understand what kind of sandboxing overhead will be incurred by applications. 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[1]. The remote write, modeled after that of Thekkath et al. [56], 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 sandboxing factor for the trusted ASH to be 1.3–1.4 for 40-byte writes, and 1.01–1.02 for 4096 bytes. 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.

As Table 6-3 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), even the sandboxed version of the specialized remote write uses fewer instructions than the generic hand-crafted one.

6.3 Exploiting ASHs and upcalls: microbenchmarks

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

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. 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). We report the *sandboxing factor* for many of the applications. We

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

define the sandboxing factor to be the time the sandboxed handler takes, divided by the time the unsafe handler takes.

Just as applications are given the entire message to process, after a demultiplexing step based on virtual circuit identifier, so 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 [23], demultiplexing of a message to an ASH was done through DPF (see Section 6.1); 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'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 [23]. 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 microseconds; under Ultrix4.2 this difference would be more like 95 microseconds (the approximate cost of an exception plus the system call back into the kernel) [23].

6.3.1 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 [13, 16]. To address this bottleneck, applications must be able to direct message placement, and to exploit ILP during copying. We examine each below.

Avoiding message copies

Message copies cripple networking performance [1, 13, 56]. 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 [33]) 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.

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

Table 6-4. 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.

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 6-4 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 also observe this effect in the 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 6-2.

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.

Integrated layer processing

The performance advantage of ILP-based composition is shown in Table 6-5, 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 3-3

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

Table 6-5. Throughput of integrated and non-integrated memory operations, measured in megabytes per second.

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

Table 6-6. 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. Two cases are considered: when the application process is running and polling for a message when the message arrives, and when the application process is suspended waiting for a message when the message arrives.

for *copy & checksum* and the composition of the checksum pipe and a byte swapping pipe, composed as shown in Figure 3-2 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.

6.3.2 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 higher cost of application upcalls.

In Table 6-6 we measure the effects of ASHs and upcalls on raw roundtrip times for a simple remote increment message. In response to the message, the application (either at user-level or in an ASH or upcall handler) receives a 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. 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 microseconds) 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 (39 microseconds), 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 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 at all, 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 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.

6.3.3 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 [61]: an efficient, unprotected transfer of control to the application in the interrupt handler.

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 microseconds to the 65-microsecond roundtrip latency [60]). 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 [56]. 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 thesis 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 Figure 6-2, 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. The slope of this line is approximately equal to the time for a single roundtrip message. This effect occurs because multiple messages are sent between the two processes when they are both scheduled; as soon as one becomes descheduled, no more messages will be sent until they are both scheduled again simultaneously. When there is exactly one other active process on the client, half of the client's time is spent processing messages and half running the other process; because all of the server's time is allocated to the server, this results in the average roundtrip time increasing by two.

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. (When the number of active processes is four, the ASH time peaks at 168 microseconds, consistently. As the time drops back down to 152 microseconds at six active processes, we assume that this represents a strange interaction between the two processors and not a fundamental problem.)

Ultrix uses a more sophisticated scheduler that raises the priority of a process immediately after a network interrupt. As Figure 6-2 shows, this type of scheduler definitely reduces the measured effect, but it is certainly still a problem.

Even when the destined process is running and polling the network, ASHs can still provide benefit. As shown by the remote increment experiments of Table 6-6, the use of ASHs still provided great benefit (a savings of 30 microseconds off the roundtrip time), 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).

6.4 Exploiting ASHs and upcalls: applications

This section reports on the performance of several applications which use ASHs, upcalls, and normal user-level communication. Specifically, it considers TCP performance, several parallel applications using the CRL software DSM system, and a Web server. Together, these applications allow us to evaluate the performance of ASHs and upcalls in a full system environment.

Note that the user-level communication measurements in this section (except for the CRL

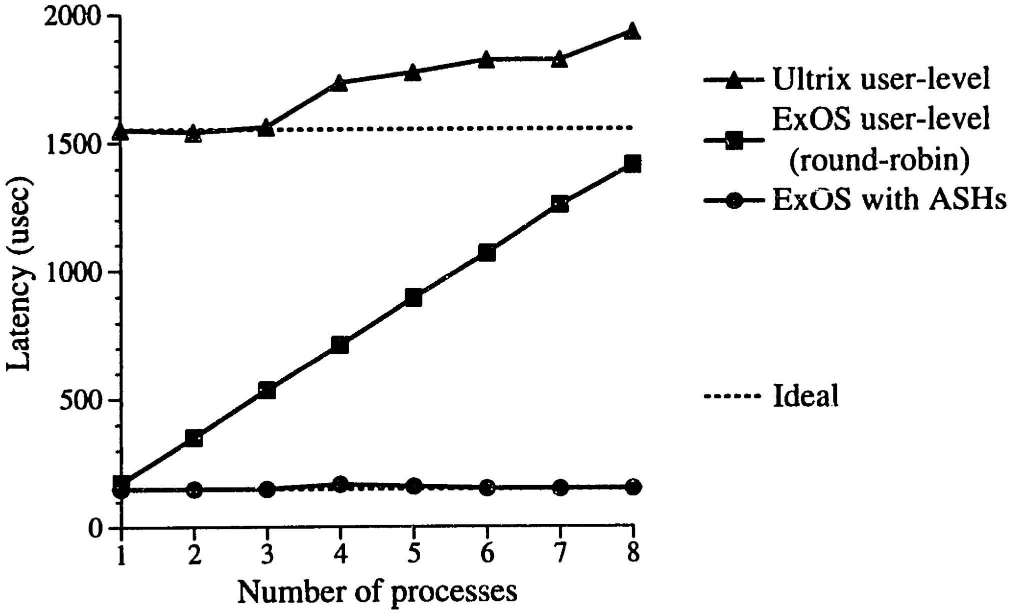


Figure 6-2. As the number of processes on the system increases, the cost of waiting for a process to be scheduled becomes increasingly higher; times are in microseconds per round trip.

ones) are taken for two cases: where the application is running and polling for the message (*polling*) and where it suspended waiting for a message and is rescheduled upon message arrival as described in Section 6.1.3 (*interrupts*). The ASH and upcall measurements are only taken when the application is scheduled; as we have shown in Section 6.3, this should not have much of an effect on the run time of the applications.

6.4.1 TCP

The TCP protocol was described in Section 6.1.5. Because TCP is important, well documented, and widely used, we illustrate the benefits of ASHs using TCP. Also, as pointed out by Braun and Diot [8], it is important to evaluate ILP in a complete protocol environment, and TCP can benefit from the use of ILP.

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

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

Table 6-7. This table compares the latency (in microseconds) and throughput (in megabytes per second) for TCP running on the AN2 for a variety of cases.

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 6-7, the use of sandboxed ASHs enables a 65 microsecond 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 microseconds slower. We expect that the performance of our sandboxer can be improved (we measure the sandboxing factor for the handler to be about 1.3). 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 microseconds difference to be more typical. The upcall performance is slightly better than that for the user-level application polling.

For this latency experiment, there is a limit on the performance that the handler versions can achieve, because 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 6-2, and shows that the use of dynamic ILP in sandboxed ASHs enables a 0.4 MBytes/second gain in throughput when the applications are not scheduled, and a 0.2 MByte/second gain in throughput when the applications are scheduled, providing performance similar to that of the *in place, with checksum* experiment of Table 6-2. The upcall version also benefits from DILP, achieving a .16 MBytes/second 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 non-local size is normally only 536 bytes [52]. 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 6-7, 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.

In summary, we have shown that although ASHs with no sandboxing overhead outperform plain user-level TCP for all cases, as we would expect, with the cost of sandboxing added the polling version of TCP outperforms the ASH ones in the latency case. The upcall case does better, because it has no sandboxing overhead. Compared to the interrupts version, both ASHs and upcalls do much better (ASHs are about 15% faster than the interrupts version, upcalls about 17%). The use of dynamic ILP on large messages both enables the integration of TCP data manipulation steps and reduces the amount of sandboxing that needs to be performed, bringing the sandboxed ASH and upcall performance up over the user-level case. Finally, when there is more work that can be done by handlers, the handler versions do even better.

6.4.2 Parallel applications using CRL

Distributed shared memory (DSM) is a programming model which provides applications with a shared address space abstraction. The C Region Library [30] is an *all-software* distributed shared memory system developed at MIT and in use by several groups outside of MIT. Because it is entirely implemented in software, it is easy to port to new platforms, and was thus a good choice for a DSM library for Aegis. The unit of coherence in CRL is called a *region*, the size of which is selectable on a data structure by data structure basis. We study three applications that use the CRL library: the Traveling Salesman Problem, Water, and Barnes-Hut.

In order to port CRL to Aegis, we wrote a very simple active message layer with similar functionality to that provided by Thinking Machines' CM-5 [34, 61] and layered this code directly above the AN2 interface. Handlers are used to perform the protocol coherence actions. The handlers are designed to fail only if interrupts are disabled (disabling interrupts is the usual mechanism used to provide atomicity for CRL implementations). This means that the parallel applications themselves did not use handlers, they merely benefited from them.

CRL is a purely interrupt-driven library. Although CRL polls while satisfying an application request, it depends on interrupts to transfer control to it if the application is executing and a message arrives. Unfortunately, there is no mechanism to provide asynchronous notification to running processes currently existing in Aegis; instead, the "interrupt queue" is checked every time a process is re-scheduled (every 15625 microseconds in our experiments). This puts the purely user-level communication version of the CRL library at a huge disadvantage, because ASHs and upcalls both can "interrupt" a running process, so the user-level communication numbers shown here are just useful for reference points. Because the ASH and upcall handlers can fail (when they cannot obtain a lock), they are backed up by the interrupt mechanism: a handler that fails to execute as an ASH or as an upcall will be placed in the interrupt queue, and offered to the application in an interrupt handler the next time the interrupt queue is checked (assuming the application has not received the message through polling by then). The interrupt mechanism itself is backed up by the application polling (*i.e.*, if the library is in a critical section when an interrupt arrives, it has disabled interrupts, and will only receive the message later when it ends the critical section and polls for the message).

Also note that because CRL uses indirect jumps, we cannot run ASHs in the kernel with sandboxing disabled (otherwise the handler would jump to the wrong code[2]), so only sandboxed numbers are reported here. The experiments reported in this section therefore cover three versions of the CRL library: normal user-level (the applications are running when a message arrives much of the time, but may not be actively polling for a message), ASH, and upcall.

As mentioned at the beginning of this chapter, our four-processor setup consists of two 40-MHz machines and two 25-MHz ones. The one- and two-processor experiments reported on here use the 40-MHz machines, but the three- and four-processor experiments also use the 25-MHz ones, so even if the applications were perfectly parallelizable and there was zero cost to communication, we would not expect to see perfect speedup in this environment. Instead, the time on three processors should be at best 1/2.625 the time of the time on one processor, and on four 1/3.25 the time.

Traveling Salesman Problem

A Traveling salesman problem (TSP) algorithm finds the shortest Hamiltonian tour of a set of cities. Our implementation uses a master/slave style program based on a branch-and-bound algorithm. The master (server) generates partial routes and stores them in a shared job queue. Each slave (client) repeatedly performs the following steps: it reads the next partial route from the job queue and then generates all full routes from the partial route by using the "closest-city-next" heuristic. All slaves keep track of the shortest route that has been found, using a shared region, which is used to prune the search tree.

We show the results for a 12 city problem, in which the master creates 990 jobs of partial length four. For this application we measure the entire search time, running the application 10 times and averaging them[3].

Figure 6-3 shows the performance of TSP on different versions of the CRL library as the number of processors increases. TSP on a single node does not communicate at all; as expected, all three versions of the library provide near-identical performance. TSP on two nodes also does not communicate, once the master node generates all of the jobs, because there is only a single slave processor to which all copies of the data migrate.

On three and four nodes, there is communication. The performance for ASHs and upcalls

[2]Because the handler has been relocated from user space to the kernel, it has no way of knowing where the relocated code has been moved to.

[3]Note that one anomalous data point (two-processor upcall) was thrown out because it was twice as large as the other data points.

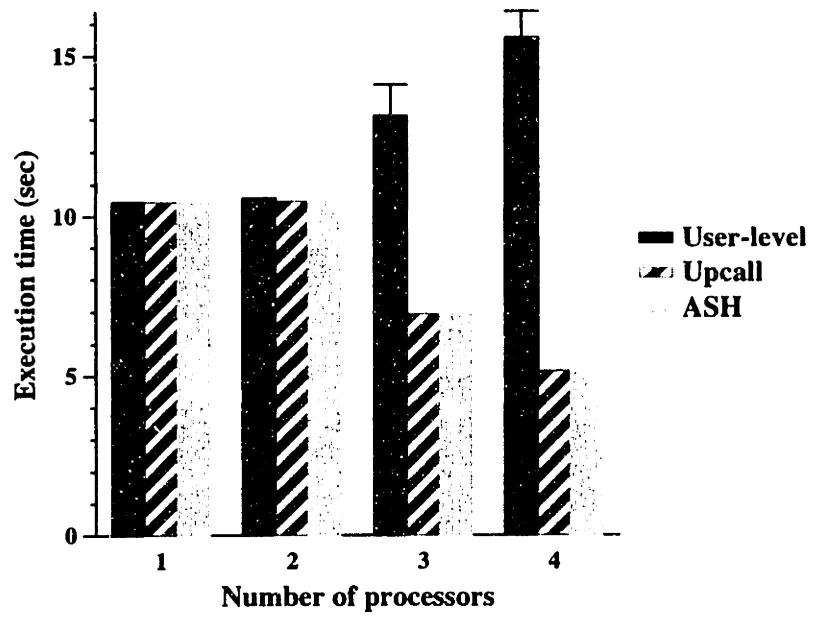


Figure 6-3. This figure shows the performance for the Traveling Salesman Problem using different versions of the CRL library as the number of processors increases. The time is measured in seconds.

is nearly identical. As explained earlier, without ASHs and upcalls CRL is impractical on the current implementation of Aegis. Note that TSP achieves a speedup of about 1.5 on three processors (the best we could expect is 1.625) and 2 on four processors (as opposed to 2.25). The confidence intervals are large for the user-level CRL library version because exactly when a message arrives can have a huge impact on the performance.

Table 6-8 shows a detailed breakdown of the success and abort statistics for the handlers and interrupts used by TSP. As expected, ASHs and upcalls have similar success rates. This rate is almost certainly good enough to make the overhead of initiating the handlers in order to test whether or not they will succeed worthwhile.

Water

Water is an n-body molecular dynamics application that "evaluates forces and potentials in a system of water molecules in the liquid state," as reported in the SPLASH parallel application suite description [49]. The version of the application that we used is adapted from the "n-squared" version from the SPLASH-2 benchmark suite [66], and is identical to that reported on in [29]. As described in [29], there is a region for each molecule and three small regions used to calculate running sums updated every iteration by each processor. The problem size that we use is 512 molecules.

As suggested in the benchmark notes, the application was run for three iterations, and the times for the second and third iteration were averaged. We then averaged 10 such runs to

# Procs	Type	Handlers				Interrupts	
		total	succ	% succ	failed	failed	succ
2	User-level	-	-	-	-	21	8
2	Upcall	59	31	52%	28	6	0
2	ASH	59	31	53%	28	0	0
3	User-level	-	-	-	-	252	1326
3	Upcall	2480	1810	73%	670	72	2
3	ASH	2280	1655	72%	625	66	2
4	User-level	-	-	-	-	824	1910
4	Upcall	3129	2154	69%	975	148	7
4	ASH	3138	2142	68%	996	164	7

Table 6-8. This table shows the abort and success statistics for the TSP application on the three versions of the CRL library averaged over the 10 runs. The first data super-column contains statistics on the number of handler invocations. *total* refers to the number of handler invocations made, which should be equal to the number of messages which arrived (summed over all processors). *succ* is the number of ASHs or upcalls that succeeded; the percentage is in the next column. The number that failed follows. Nearly all of the failures were due to "interrupts" being disabled when the message arrived. The second data super-column contains statistics on the number of interrupts run. Note that a failure may represent multiple messages being blocked from running (it is incremented with every interrupt invocation), but the *succ* column counts every message that was handled in an interrupt handler (not as an ASH nor upcall).

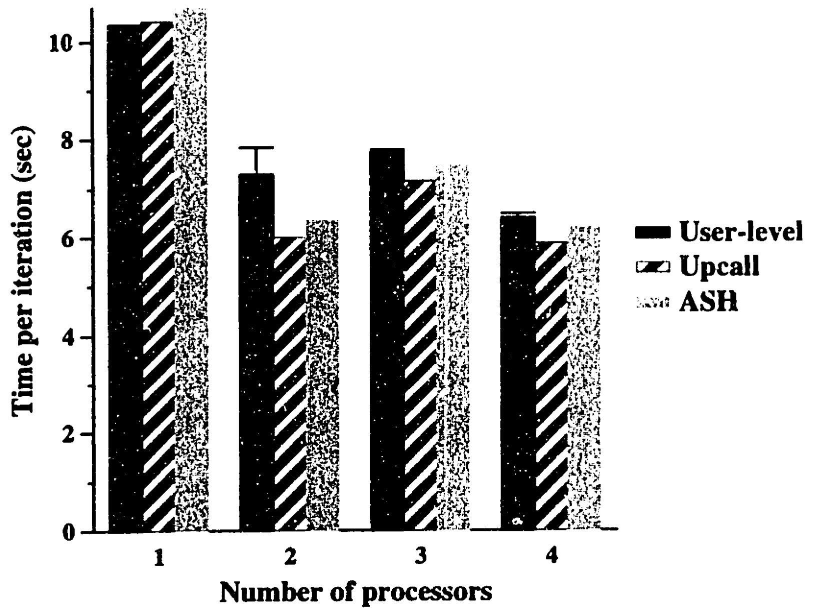


Figure 6-4. This figure shows the performance for the Water application using different versions of the CRL library as the number of processors increases. The time is measured in seconds.

calculate the data presented here.

There is much more communication relative to the amount of computation in the Water application than there is in the Traveling Salesman Problem application. Additionally, the Water application is structured so that the application is often waiting in the CRL library (and therefore polling); in fact, unlike Barnes-Hut and TSP, this application will run correctly with no form of interrupts enabled even if no polling calls are manually added to the application. This explains why the performance of Water for the normal user-level communication version of the CRL library achieves the level of performance that it does. Since there is no communication during the one-processor experiment, all versions should take the same amount of time, however the ASH version is slightly slower. This may be a cache conflict type of problem, as was mentioned in Section 6.1.2.

As the number of processors increases, the ASH version does better than the user-level version. The upcall version does best, however. None of the versions achieves good speedup; the overhead for communication on our hardware is too large to support this level of granularity.

Table 6-9 shows a detailed breakdown of the success and abort statistics for the handlers and interrupts used by Water. The success rate is lower than that for TSP. The fact that the failures are nearly all due to interrupts being off suggest that an additional mechanism would be useful: applications should be able to turn on and off ASHs (and upcalls) in a way that the operating system can test for before invoking the handler (similarly to the way that applications can turn on or off asynchronous notification in a normal operating system). This strategy would greatly decrease the penalty for initiating handlers that will not be able to complete due to application-dependent, as opposed to message-dependent reasons, and should therefore

# Procs	Type	Handlers				Interrupts	
		total	succ	% succ	failed	failed	succ
2	User-level	-	-	-	-	4157	1619
2	Upcall	20963	8283	39%	12680	2980	8
2	ASH	20823	8461	40%	12362	2905	1
3	User-level	-	-	-	-	9513	2577
3	Upcall	33949	14501	42%	19449	6516	46
3	ASH	33786	13760	40%	20026	6294	79
4	User-level	-	-	-	-	14985	3036
4	Upcall	47296	19801	41%	27495	10732	85
4	ASH	47577	20838	43%	26739	10081	111

Table 6-9. This table shows the abort and success statistics for the Water application on the three versions of the CRL library averaged over the 10 runs. The first data super-column contains statistics on the number of handler invocations. *total* refers to the number of handler invocations made, which should be equal to the number of messages which arrived (summed over all processors). *succ* is the number of ASHs or upcalls that succeeded; the percentage is in the next column. The number that failed follows. Nearly all of the failures were due to "interrupts" being disabled when the message arrived. The second data super-column contains statistics on the number of interrupts run. Note that a failure may represent multiple messages being blocked from running (it is incremented with every interrupt invocation), but the *succ* column counts every message that was handled in an interrupt handler (not as an ASH nor upcall).

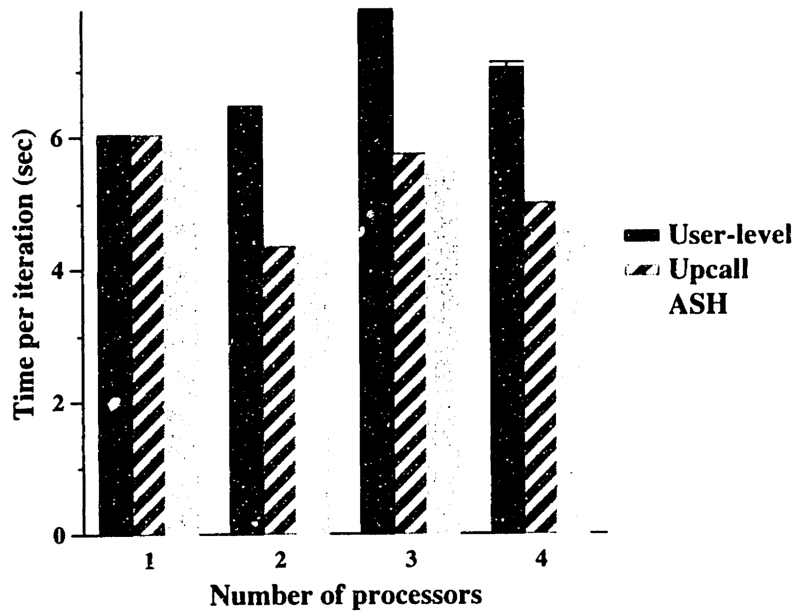


Figure 6-5. This figure shows the performance for the Barnes-Hut application using different versions of the CRL library as the number of processors increases. The time is measured in seconds.

increase both application and system performance as a whole.

Barnes-Hut

Barnes-Hut is also taken from [29], and also originated as a SPLASH-2 application. Barnes-Hut "simulates the evolution of a system of bodies under the influence of gravitational forces" [29] using hierarchical n-body techniques. As described in [29], there is a region for each of the octtree data structure elements present in the SPLASH-2 code: bodies, tree cells, and tree leaves. The global sums, minima, and maxima use the CRL reduction primitives, which are built directly on top of the active messaging system used to implement CRL on Aegis. The problem size that we used is 1024 bodies.

The steady state behavior for this application is achieved only after the second iteration, so the application was run for four iterations, and the times for the third and fourth iterations were averaged. Ten sets of these averages were used to calculate the data presented here.

Barnes-Hut has an even smaller granularity of computation than Water. Furthermore, it relies on interrupts to operate correctly, and will not complete if interrupts are disabled. The performance of this application running on top of the user-level CRL version is thus abysmal; adding more processors actually slows down the total running time in most cases because the hardware overhead is so large. Both the ASH and the upcall version of the CRL libraries provide better performance, with the ASH version having a slight edge in the two- and three-processor case (in the four-processor case, the ASH version is slightly better, but the error bars overlap).

# Procs	Type	Handlers				Interrupts	
		total	succ	% succ	failed	failed	succ
2	User-level	-	-	-	-	8655	1502
2	Upcall	32224	13849	43%	18375	4467	20
2	ASH	32221	14312	44%	17909	3964	10
3	User-level	-	-	-	-	17285	2629
3	Upcall	56671	26049	46%	30622	9746	63
3	ASH	56778	26172	46%	30606	9148	37
4	User-level	-	-	-	-	27718	4413
4	Upcall	82811	37415	45%	45396	15965	114
4	ASH	82965	37852	46%	45113	15195	96

Table 6-10. This table shows the abort and success statistics for the Barnes-Hut application on the three versions of the CRL library averaged over the 10 runs. The first data super-column contains statistics on the number of handler invocations. *total* refers to the number of handler invocations made, which should be equal to the number of messages which arrived (summed over all processors). *succ* is the number of ASHs or upcalls that succeeded; the percentage is in the next column. The number that failed follows. Nearly all of the failures were due to "interrupts" being disabled when the message arrived. The second data super-column contains statistics on the number of interrupts run. Note that a failure may represent multiple messages being blocked from running (it is incremented with every interrupt invocation), but the *succ* column counts every message that was handled in an interrupt handler (not as an ASH nor upcall).

Number of requests per connection	Number of iterations
1	1000
5	1000
10	1000
100	1000
10000	10

Table 6-11. This table shows the parameters used in the Web server experiment.

Table 6-10 shows a detailed breakdown of the success and abort statistics for the handlers and interrupts used by Barnes-Hut. The success rate is slightly better than that for Water, but still significantly worse than TCP.

Summary The CRL applications demonstrate that having efficient asynchronous notification in a system is crucial. They also show that sandboxing did not have a significant negative effect for these applications: the performance of ASHs and upcalls is very similar. If the sandboxing overhead were lower, we would expect the performance for ASHs to increase. As mentioned earlier, the ability for applications to turn on and off ASHs and upcalls in an operating-system-visible way would increase the performance for both ASHs and upcalls, as time would not be wasted initiating handlers that immediately (*i.e.*, as soon as they examine application state) abort. Our implementation placed only CRL library code in handlers. One could imagine integrating the CRL code with the applications themselves, and placing the result in handlers; we expect that this type of tight coupling would provide performance benefits, but at a cost to the programmer (at the very least, it would require changes to the CRL interface).

6.4.3 A Web server

A Web server is an example of a widely used client-server application. The World Wide Web is basically a repository of information distributed around computing sites all over the world. Web servers store *pages*. Web clients can request pages from servers using a Uniform Resource Locator (URL). The URL specifies which server has the data, and where on the server it is located.

Web servers and clients communicate using the HTTP protocol, which runs on top of TCP. Clients request data by opening a TCP connection to a server and requesting a URL; servers read the request, and respond with the requested data. At this point, the connection is closed (either by the client or the server). Each new request requires a new connection to be opened. Because of the inefficiency of this system (since the same client often issues multiple requests to the same server), proposals have been made to allow connections to stay open across multiple requests [39]. We examine both single-request and multiple-request connections here.

In our experimental setup, the client opens a connection and then repeatedly performs the

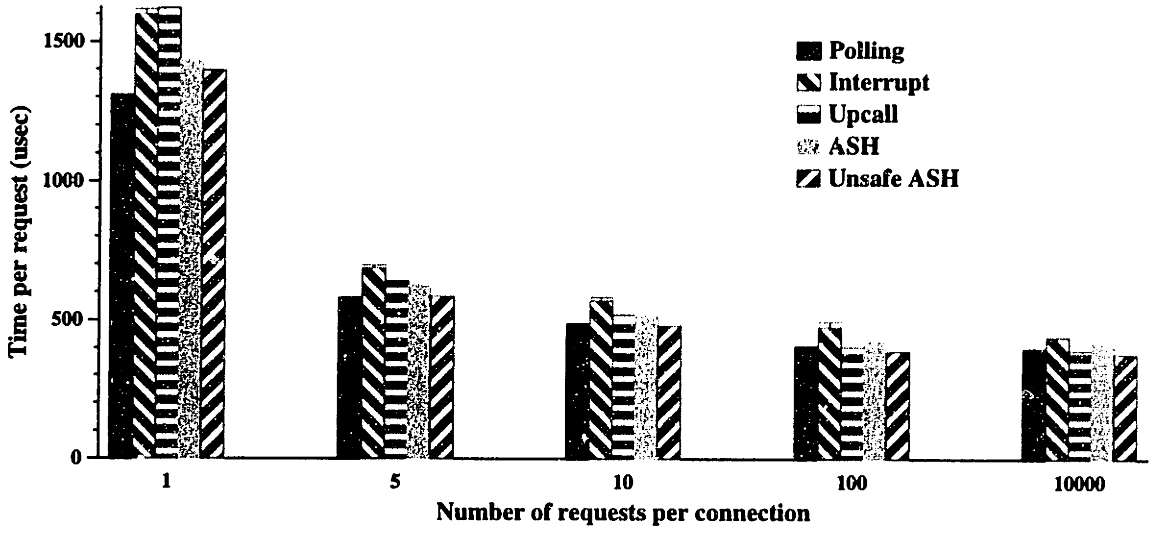


Figure 6-6. This figure shows the performance of multiple versions of the Web server application as the number of requests per connection increases. The time is measured in microseconds per page requested. The page size for this experiment was eight bytes.

following steps: (1) it sends a request to the server (2) it checks if there is a response, and if so reads it. It does this up to the number of requests per connection allowed for that particular data point (as shown in Table 6-11). The client then reads data until it has received the responses to all of its requests. This loop is repeated multiple times (exactly how many times varied with the number of requests per connection, see Table 6-11). The time for this entire process was measured, then divided to calculate the time per request. Each data point was measured in 10 separate runs, which were then averaged.

All of the requests are to the same page, and furthermore this page is always assumed to be in the disk cache in our setup. This represents the best possible case (*i.e.*, where network performance and software overhead matter the most). We consider two page sizes in our experiments: the first is eight bytes, which is smaller than any page would likely be, but represents the case where the software overhead should matter the most, and the second is 2048 bytes, which was estimated to be approximately the median size for pages by Padmanabhan and Mogul [44].

The handler used for both ASHs and upcalls in this set of experiments is built on top of the one for TCP discussed in Section 6.4.1. It handles the low-level TCP protocol actions without invoking Web-server-specific code (for example, the sending of acknowledgments) to certain control messages. If there is data to be given to the Web server, the Web server portion of the handler invoked. This code looks up the URL in a simulated disk cache (using the lookup code taken from our real Web server), and if the page is there sends the response out immediately.

Figures 6-6 and 6-7 demonstrate similar results for both page sizes. As expected, the difference between the different communication strategies is smaller for the 2048-byte pages.

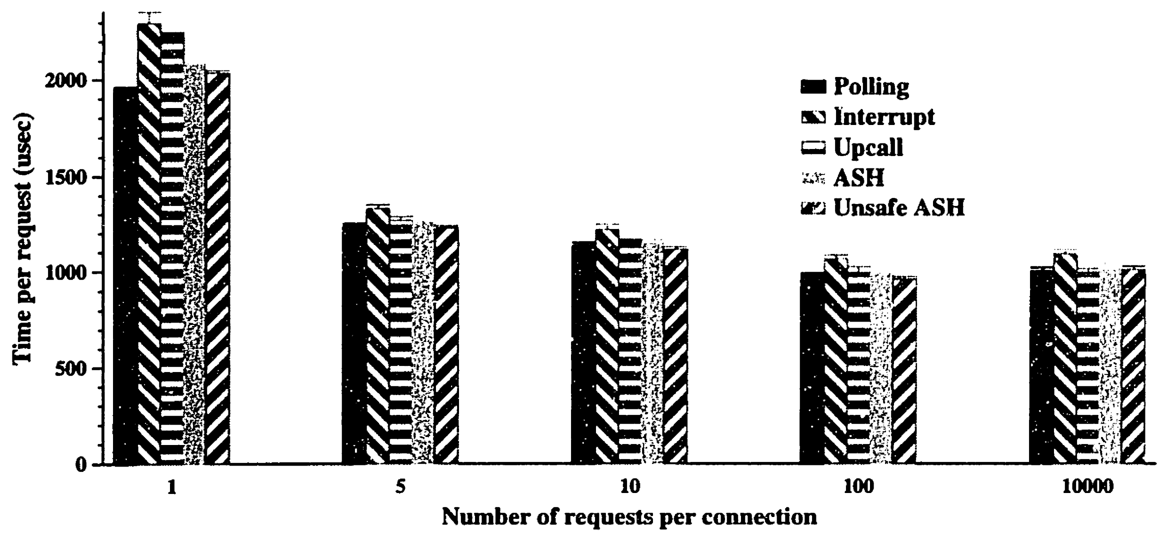


Figure 6-7. This figure shows the performance of multiple versions of the Web server application as the number of requests per connection increases. The time is measured in microseconds per page requested. The page size for this experiment was 2048 bytes.

There are several interesting effects to note. First, when there is only a single request per connection, both ASHs and upcalls are significantly slower than polling. This is because of the high failure rate for them: 70-80% of them fail outright (since the handlers do not handle TCP setup and teardown, because they are built on top of the TCP handler which only handles the "header predicted" cases). For each one that fails, the time to initiate the ASH (upcall) and the time to clean up after it are wasted; since the overhead for initiating an upcall is greater than that for initiating an ASH, upcalls are penalized more when handlers fail quickly.

As the number of requests per connection increases, the percentage of successful handlers increases, because the TCP header prediction code only rejects opening and closing messages. There is thus a greater opportunity for handlers to succeed. Table 6-12 shows the percentage of successful handlers as a function of connection size (this was measured for ASHs, but the data should be nearly identical for upcalls). It shows the success rate both for the TCP handler alone and for the server application (*i.e.*, some messages contain no data for the server at all, and can be entirely handled by TCP). It also shows the best possible success rate, assuming that there were no lock conflicts between the application and the handlers. This is calculated by a simple formula. A connection consists of $r + 4$ total messages arriving at the server, where r is the number of requested documents, and 4 is the number of TCP setup and shutdown overhead messages. Of these messages, at most r can reach the server, so the best server success rate is $r/(r + 4)$. It turns out that TCP can predict one of the 4 control messages, so the best success rate considering just the TCP handler is $(r + 1)/(r + 4)$. The reason that the actual success percentage is lower for smaller numbers of requests per connection is that many of the handlers fail due to lock collisions between the server application (and its TCP library) and the handler

Requests per connection	% Successful	Best possible	% Successful (application)	Best possible (application)
1	20-30%	40%	10-15%	20%
5	55-60%	67%	50-55%	56%
10	70-75%	78%	65-70%	71%
100	96-97%	97%	95-96%	96%
10000	100%	100%	100%	100%

Table 6-12. This table shows the success rate statistics for the Web server experiment. For each number of requests per connection, it records the percentage of successful ASHs and upcalls (out of all tried), the upper bound on the percentage, the percentage of successful ASHs and upcalls that performed Web server action (as opposed to just TCP action), and the upper bound on that percentage.

(and its TCP library portion).

At greater numbers of requests per connection, the polling, not sandboxed ASH, and upcall versions all have similar times. The sandboxed ASH version is more expensive (although still cheaper than the interrupt version), since the sandboxing overhead matters. It is interesting to note that the sandboxing overhead is much larger for the smaller page size (the sandboxing factor for the ASH handling eight-byte pages is 1.3 (*i.e.,* the sandboxed handler takes 1.3 times as long to run as the one that is not sandboxed), but only 1.1 for 2048-byte pages). This effect occurs because in the 2048-byte case more time is spent in the handler processing message data (calculating the TCP checksum) than in the 8-byte case. Since the checksum calculation is done using the loop generated by our DILP engine (and not user code), the calculation does not need to be sandboxed, therefore a large percentage of the total handler time represents code which is not sandboxed.

6.5 Summary and discussion

This chapter demonstrated that although the performance of user-level networking alone on the exokernel is good, through the selective use of ASHs and upcalls it can be made better. The microbenchmarks demonstrate the isolated benefits of being able to run application code in response to a message: high throughput, low-latency data transfer, and low-latency control transfer. We also used them to address the issue of performance when there is more than one active process using the machine, showing that ASHs and upcalls deliver performance consistently better than user-level communication alone when there are multiple processes.

The applications that we presented give us insight into the ways handlers can be used in different application domains. The TCP protocol demonstrates that dynamic ILP could be both used and useful; the combined checksum and copy definitely increased the throughput we could deliver. The parallel applications running on top of CRL show that ASHs and upcalls can successfully be used to deliver efficient asynchronous notification to running applications even when the normal operating system does not provide this capability and also demonstrate the flexibility available on our system. With no changes to the applications or the kernel, we can replace the coherence protocol of CRL. They also suggest that an additional mechanism for applications to be able to enable and disable handlers would be beneficial. Finally the Web server application shows how the performance of the system degrades when the percentage of aborts is too high: both ASHs and upcalls perform poorly, but ASHs perform better because they are cheaper to initiate and abort. Disabling handlers from the application would be useful for a Web server on a much coarser granularity than for the CRL applications: if the server noticed that none of its connections were staying open for longer than one document, it would want to disable the handlers.

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 implementation effort has been focused on correctness rather than performance, and should be able to be improved. For example, normal procedure returns are treated as general-purpose register-indirect jumps currently. For procedures called from a single call site, the return can be replaced with a jump to the correct location. 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* (PCC) [43] may provide the ability to download complex kernel code with little to no extra runtime overhead.

Additionally, 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 actually performed this experiment. Liedtke, a researcher who has written an extremely high performance implementation of upcalls, believes that the only way to do so is to completely rewrite the operating system from scratch [36]. Finally, we strongly believe that it is easier to implement ASHs (or even upcalls) into an existing OS (*e.g.*, NT) than it is to change an existing OS into a super fast one (*e.g.*, Aegis or L4).

Chapter 7

Analysis

In the last chapter, we saw that both ASHs and upcalls provide the ability to run code immediately in response to a message. We also noticed that there are a number of tradeoffs between using these different models of communication. For example, ASHs are more tightly coupled with the operating system than are upcalls, and thus the overhead for system calls can be avoided when invoking kernel functions from an ASH. ASHs also have a smaller overhead for initiation than ASHs do, because upcalls have to cross a protection boundary (from in the kernel to outside the kernel, and then back). On the other hand, ASHs incur some amount of overhead due to making them safe, which does not occur for upcalls.

In order to gain a better understanding of when these tradeoffs matter, and furthermore to expand our study to other architectural models and operating systems, we developed an analytical model to show which communication technique performs best under what circumstances. For a set of operating system and hardware parameters, we are interested in varying three parameters: the length of the handler used by the application, the sandboxing overhead for that application, and the probability that the handler will abort.

In this chapter, we develop this model. We then perform a study of two operating systems running on a single architecture: Aegis and an Ultrix-like operating system running on the hardware platform used for our experiments. We then look at the behavior of normal user-level communication, upcalls, and ASHs for different application characteristics.

7.1 Assumptions

The model was designed to be fairly general. We do assume that a hardware interrupt is generated by the processor in response to a message, and thus that the processor is in the device driver of the operating system at the time we start our measurements. This is not true of all devices. In particular, network interfaces which have been specifically designed for user-level communication may not need to generate an interrupt at all upon message arrival (*e.g.,* Hamlyn [9]). For those types of systems, handlers are not useful if the application is running during message arrival. On the other hand, if the application is not running, some sort

Process state code	**Mnemonic**	**Process status**
rp	spinning	Running and actively Polling for a message
rb	busy	Running but Busy executing other application code
sw	blocked	Suspended and Waiting for a message
sb	busy-suspended	Suspended and not waiting for a message (Busy)

Table 7-1. The codes used to represent process state in the model.

of scheduling event should occur to enable timely responses; an ASH or upcall could be the appropriate mechanism for this.

Another assumption that is made by the model is that messages do not need to be copied out of the network interface immediately. If they do, then the penalty of not using handler events should be increased (because the system should have to move the message data out of the NI itself, quite possibly to the wrong location, in order to keep the NI free). An example of an interface that requires such quick action is Ethernet. Ignoring this effect makes ASHs and upcalls look comparatively worse than they should be.

One thing that the model does not model explicitly is handlers that have sandboxed and not-sandboxed parts: for example handlers that do some amount of general-purpose computation and then a fair bit of data motion using loops generated by the DILP compiler. For simplicity, we model this effect by picking a sandboxing factor which represents the combination of these two handler parts, but one could easily specify a model in which the handler run time was split into a sandboxed and non-sandboxed parts, and in which the non-sandboxed run time was parameterizable.

In Table 7-1 we consider the four possible states that a process could be in. The first is *spinning*, for when an application is running and is actively polling for a message. This state corresponds to the state called *polling* in the previous chapter. The second is when an application is *busy* running but is not polling for a message. This state occurs for our parallel applications running on top of CRL (as described in Section 6.4.2), when the application is actively executing and expecting messages to be handled in the background. The third case is when an application is suspended and is waiting for a message; we refer to this case as *blocked*. The assumption here is that the operating system will increase the priority of this process and reschedule it when a message arrives (this is not true for all operating systems, but those for which it is not can be modeled by the fourth state). The final state is if an application is suspended and is not waiting for a message (*busy-suspended*); this would occur if an application in the second state became suspended due to using up its time slice, or perhaps waiting on a non-network resource.

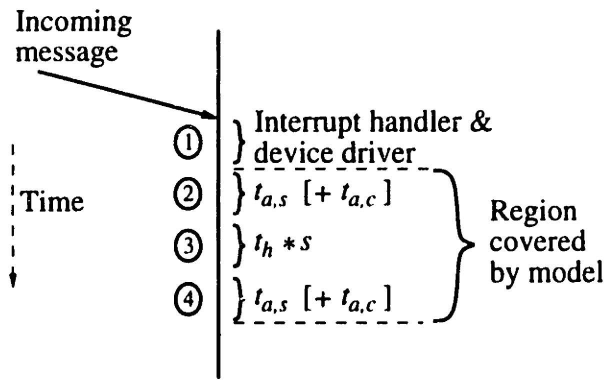


Figure 7-1. This figure shows the events that occur in response to a message which is handled completely by an ASH, and illustrates which ones are covered by the model. The first step that occurs after a message arrives is that the processor is interrupted, and the interrupt handler and device driver are run. At a particular point in the device driver execution, ASHs may be run; this is the point the model starts measuring from, and is the start of Step 2 in the diagram. Step 2 involves setting up the ASH to run, and doing whatever is needed to switch to the right application if it was not already running. Step 3 involves running the ASH itself, and finally Step 4 covers the time to clean up after the ASH completion and, if necessary, switch back to the application that was running before the ASH was invoked.

7.2 Model

We want to provide a basis to evaluate the differences between unaugmented user-level communication, upcalls, and ASHs; we therefore build a model of each of these communication strategies. We start by describing the parameters used by the model, then we describe the model itself. Figure 7-1 gives an idea of how the parameters are used in the model by relating the model parameters to one set of events that might occur in response to a message (a successful ASH invocation); Figure 7-2 shows a different set of events (the ASH aborts, and then the message is delivered to the suspended application process via an interrupt).

7.2.1 Parameter description

There are a large number of machine- and operating-system-dependent parameters used by the model, as shown in Table 7-2. The first set of parameters shown in the table pertains to ASHs, and represents the time to set up an ASH to run, the time to do an address space switch (*i.e.*, what needs to be done to avoid a full context switch) if the application that the message belongs to was not the application that the message is for, the time to clean up after a successful ASH and the time to clean up after a voluntary ASH abort. There is a similar set of parameters for upcalls. For interrupts (which will have an effect on the application when it arrives even if it is running), we have just the time to do the interrupt and switch to the interrupt handler

Parameter	Applies to	Represents time to...
$t_{a,i}$	ASH	set up an ASH to run (assuming process is scheduled)
$t_{a,c}$	ASH	"context switch" to run ASH (if necessary)
$t_{a,a}$	ASH	clean up after voluntary ASH abort
$t_{a,s}$	ASH	clean up after successful ASH completion
$t_{u,i}$	upcall	set up an upcall to run (assuming process is scheduled)
$t_{u,c}$	upcall	"context switch" to run upcall (if necessary)
$t_{u,a}$	upcall	clean up after voluntary upcall abort
$t_{u,s}$	upcall	clean up after successful upcall completion
$t_{i,i}$	interrupt	set up & switch to a process (assuming it is scheduled)
t_c	interrupt	do a full context switch to a process
$t_{p,i}$	poll	return to a running process which is polling
t_q	-	time quantum

Table 7-2. The base machine- and OS-dependent parameters used in the model.

Parameter	Applies to	Meaning
t_h	-	time to run a handler (not sandboxed)
s	ASH	slowdown factor due to sandboxing
$p_{a,a}$	ASH	probability that an ASH will abort
$p_{u,a}$	upcall	probability that an upcall will abort
f	handler	fraction of the handler that will run if handler aborts
n_p	-	number of other active processes

Table 7-3. The application and workload-dependent parameters used in the model.

within the process and also the time for a full context switch to the process (if necessary). For polling, there is just the time to return from the kernel interrupt to the polling process. The last parameter is the time quantum that processes are scheduled at: this parameter represents how long a process will run for that is not preempted and does not voluntarily relinquish the processor.

Table 7-3 lists the parameters that depend on the application and the workload of the processor. These are the ones that we will be varying. These parameters are: the running time of the handler before it is sandboxed (t_h), the slowdown factor due to sandboxing, as defined in Chapter 6 (s), the probability that an ASH or upcall will abort ($p_{a,a}$,$p_{u,a}$), the fraction of the handler that will run before the ASH or upcall can determine that it will need to abort (f), and the number of active processes running on the processor at the time of message arrival (other than the one that the message is for).

The model calculates the values of several variables, as shown in Table 7-4. The first five are

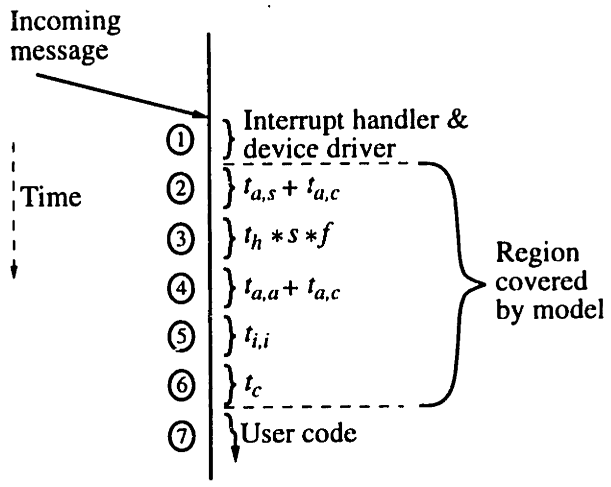


Figure 7-2. This figure shows how the model represents the case where an ASH starts running for a process that was suspended and waiting for a message, and then subsequently aborts. Steps 1 and 2 remain the same as in Figure 7-1. In Step 3, on the other hand, only part of the ASH is run before the abort, and in Step 4, the abort clean up code is run instead of the success clean up code. At this point, a full set up to run user interrupt code must occur, and the a full context switch. The model stops counting the time after this point. Finally, user code can be run. Note that in this case, the user code is at full status, not handler status.

Parameter	Applies to	Meaning
$t_{a,ps}$	ASH	time to execute an ASH (assuming process is scheduled)
$t_{u,ps}$	upcall	time to execute an upcall (assuming process is scheduled)
$t_{i,ps}$	interrupt	time to execute an interrupt (assuming process is scheduled)
$t_{p,ps}$	poll	time to execute a poll (assuming process is scheduled)
t_{ps}	-	time to schedule a not-already-running process, on average
$t_{a,xx}$	ASH	time to execute an ASH (assuming process is in state xx)
$t_{u,xx}$	upcall	time to execute an upcall (assuming process is in state xx)
$t_{i,xx}$	interrupt	time to execute an interrupt (assuming process is in state xx)
$t_{p,xx}$	poll	time to execute a poll (assuming process is in state xx)

Table 7-4. The variables derived by the model.

intermediate variables, chosen to simplify the calculation. $t_{a,ps}$ represents the time to execute an ASH assuming that the associated application process was scheduled when the message arrived. It only partly considers what happens if the ASH fails. $t_{u,ps}$ is the identical calculation for upcalls. $t_{i,ps}$ and $t_{p,ps}$ are the time to reach and then execute a handler when doing an interrupt to a running process and a poll from a running process, respectively. t_{ps} represents the time to schedule a process that is not running, assuming that the scheduler is not giving it any special priority (and is using round-robin scheduling).

The last four variables of Table 7-4 represent the outputs of the model. They are the time to execute an ASH upcall, interrupt, and poll, respectively, depending on what state the process was in when the message arrived.

7.2.2 Model description

We will build up to calculating the abovementioned output variables (there are actually sixteen of them if every combination is considered, but only thirteen make sense, as will be described below). We start by simplifying and combining the input parameters.

Since the model only considers voluntary aborts by ASHs and upcalls, the time to clean up after a successful ASH (or upcall) should equal the time to clean up after one that aborts:

$$t_{a,a} = t_{a,s}$$

$$t_{u,a} = t_{u,s}$$

We assume that the probability of an ASH aborting voluntarily is the same as the probability of an upcall aborting voluntarily.

$$p_{a,a} = p_{u,a}$$

We now calculate the time to run the four alternatives, under the assumption that the process the message is for is currently running. If the handling method is polling, then the time required is just the time to exit the kernel, return to the application, have the application notice the message, and then run the handler:

$$t_{p,ps} = t_{p,i} + t_h \tag{7.1}$$

Similarly, if the handling method is interrupts, then the time required is just the time to exit the kernel, return to the application, interrupt it, and then run the handler:

$$t_{i,ps} = t_{i,i} + t_h \tag{7.2}$$

If the handling method is upcalls, then the time required is the time to initiate the upcall, and then run the handler, and then clean up afterwards. Since only part (the fraction f) of the handler will run if the handler aborts, we have to subtract out the portion that will not run in case of failure. It is easy to see that if f is 1, the handler will be counted fully no matter what the probability of abort.

$$t_{u,ps} = t_{u,i} + t_h * (1 - p_{u,a} * (1 - f)) + (1 - p_{u,a}) * t_{u,s} \quad (7.3)$$

The calculation for ASHs is similar, only the time to execute the handler may be longer because it is sandboxed:

$$t_{a,ps} = t_{a,i} + t_h * s * (1 - p_{a,a} * (1 - f)) + (1 - p_{a,a}) * t_{a,s} \quad (7.4)$$

Finally, we calculate the time to schedule a process, assuming that the process is not scheduled, that it is not explicitly waiting on a message (so that message arrival will not change its scheduling priority) and that there is a round-robin scheduling scheme in use:

$$t_{ps} = \frac{n_p}{2} * (t_q + t_c) \quad (7.5)$$

With the above variables, we can now consider calculating the time spent by the processor with each scheme. There are four interesting cases to consider, described in Table 7-1. The first is *spinning* (rp), for when the process that the message is for is running and is actively polling the network for a message. For this case, we do not calculate a time for interrupts.

The time taken for a polling process in this case is, by definition, the time we previously calculated for a polling process when the process was scheduled.

$$t_{p,rp} = t_{p,ps} \quad (7.6)$$

The time for an upcall in this case is equal to the time for an upcall to a scheduled process, plus the probability of the upcall failing times the time needed if the upcall fails. Note that if the upcall fails, the time taken after the failure is the same as that for the polling case. The calculation for the ASH is identical.

$$t_{u,rp} = t_{u,ps} + p_{u,a} * (t_{u,a} + t_{p,rp}) \quad (7.7)$$
$$t_{a,rp} = t_{a,ps} + p_{a,a} * (t_{a,a} + t_{p,rp})$$

The second case is *busy*, or rb, for when the process is running, but is busy executing other application code. We do not calculate a time for polling in this case, because it is too dependent on the application polling characteristics.

The time taken for an interrupt-style process in this case is, by definition, the time we previously calculated for an interrupt-style process when the process was scheduled.

$$t_{i,rb} = t_{i,ps} \quad (7.8)$$

The time for an upcall in this case is nearly identical to that for the previous case (Equation 7.7). The only difference is that if the upcall fails, the additional time will come from an interrupt instead of a poll. Again, the ASH calculation is identical.

$$t_{u,rb} = t_{u,ps} + p_{u,a} * (t_{u,a} + t_{i,rb}) \quad (7.9)$$

$$t_{a,rb} = t_{a,ps} + p_{a,a} * (t_{a,a} + t_{i,rb})$$

The third case is *blocked*, or sw. In this case, the process is suspended and waiting for the message. For polling-style processes, the process was in the middle of the polling loop when it was suspended. For interrupt-style processes, we assume that the process ran out of work, and suspended itself in such a way that the scheduler will bump it to the top of the queue when a message arrives.

Because the scheduler does not know that the polling process was in the middle of waiting for a message, it will not reschedule the process until it is its turn again. This means that the time for a polling process depends on the number of other active processes in the system:

$$t_{p,sw} = t_{ps} + t_{p,ps} \tag{7.10}$$

For interrupts, on the other hand, we assume that the scheduler will increase the priority so upon message arrival, the process that the message is intended for will be scheduled immediately, and the currently-running process will be switched out. As soon as this full context switch occurs, the case reduces to the previous one (*i.e.*, that of an already-scheduled process being interrupted).

$$t_{i,sw} = t_c + t_{i,ps} \tag{7.11}$$

More overhead is incurred for an upcall in this case, as well. Although a full context switch does not need to be performed before the upcall, at least part of one must, so that the message can be handled. Furthermore, whether or not the upcall succeeds, the switch back must be accounted for (for interrupts, we assume the process continues running after receiving the message, so do not count the cost of the full context switch twice). Additionally, if this upcall fails, the cost for a full context switching interrupt must be incurred[1]. Again, the ASH calculation is nearly identical.

$$\begin{aligned} t_{u,sw} &= 2 * t_{u,c} + t_{u,ps} + p_{u,a} * (t_{u,a} + t_{i,sw}) \qquad (7.12) \\ t_{a,sw} &= 2 * t_{a,c} + t_{a,ps} + p_{a,a} * (t_{a,a} + t_{i,sw}) \end{aligned}$$

The fourth case is *busy-suspended*, or sb. In this case, the process is again suspended. This time, however, it is not explicitly waiting for a message, but instead executing other application code (this would occur if the application got swapped out while it was busy (in state rb), for example because it ran to the end of its time slice) . We again exclude the polling-style applications from this case, for the same reason as for case rb.

The interrupt case is very similar to the polling case for the previous process state situation (where the process was suspended and waiting for a message). Because the process is not actively waiting for a message, the scheduler will not know to reschedule it, and thus the process must wait its turn:

[1]Because the cost of waiting for a poll would be prohibitive, we do not bother calculating the time for an upcall or ASH for polling-style processes in this case.

Parameter	Aegis	Ultrix-like
$t_{a,i}$	5.4	5.4
$t_{a,c}$	.4	.4
$t_{a,a}$	1.7	1.7
$t_{a,s}$	1.7	1.7
$t_{u,i}$	26	26
$t_{u,c}$	.4	.4
$t_{u,a}$	4.9	4.9
$t_{u,s}$	4.9	4.9
$t_{i,i}$	22	30
t_c	24.5	75
$t_{p,i}$	20.9	30
t_q	15625	15625

Table 7-5. The input parameters used to describe Exokernel and Ultrix-like behavior on the DECstation 5000/240s used in our experiments to the model. All numbers are measured in microseconds.

$$t_{i,sb} = t_{ps} + t_{i,ps} \tag{7.13}$$

The upcall case, on the other hand, is nearly identical to the previously calculated one. The only difference is that if the upcall fails, the process must wait until it is scheduled.

$$\begin{aligned} t_{u,sb} &= 2 * t_{u,c} + t_{u,ps} + p_{u,a} * (t_{u,a} + t_{i,sb}) \\ t_{a,sb} &= 2 * t_{a,c} + t_{a,ps} + p_{a,a} * (t_{a,a} + t_{i,sb}) \end{aligned} \tag{7.14}$$

7.3 Input parameters

This section describes the values of the hardware- and operating-system-dependent input parameters of the model.

Exokernel parameters The exokernel (Aegis) parameters were measured directly on our platform. All of the numbers were measured 1000 times, using the cycle counter on the AN2 board. After examination of a histogram of the values either the average (arithmetic mean) or the minimum value was used; we will note specially which numbers were taken using the minimum values. These values are not completely precise (for example, adding counters to the kernel perturbed the run times of the experiment due to cache conflicts, and we were forced to apply the methodology described in Section 6.1.2 here). We tried to make sure any errors would penalize upcalls and ASHs, not user-level communication. Note that two successive reads of the cycle counter always returns 2 microseconds, so we subtracted that from the times reported here.

The ASH initiation and termination times were measured as follows. $t_{a,i}$ was measured from the point in the AN2 device driver where it checks whether or not there is an ASH to run to the first line of the user code at the start of an ASH. The time for a voluntary abort in our system is assumed to be the same as that for a normal termination (since they share a code path), so we just measured the time for a normal termination to determine $t_{a,s}$. This was measured from the last user code in an ASH back to the device driver. Note that after this point the device driver is not finished, and furthermore that control must return to the running process, however the message has been completely handled at this point. The upcall initiation and termination times were measured in the same way. The "context switch" time for ASHs and upcalls was assumed to be the same, because they use the same code. It was calculated by subtracting the minimum value of $t_{a,i}$ taken when there was no context switch needed from the minimum value taken when there was a context switch needed.

The time to initiate a poll ($t_{p,i}$) was measured from the same point in the device driver to the first line of user code located after a polling loop (the process was polling on an address in local memory). We report the average value after throwing out seven of the 1000 data points collected (because those seven points, two orders of magnitude above the others, were clearly erroneous).

The time to initiate application code not currently running (*i.e.*, $t_{i,i}$) was measured using our simulated interrupt mechanism described in Section 6.1.3. The minimum value (the average was much higher) was 30 microseconds. Subtracting out the time for the poll part (as that is the first part of the simulated interrupt) gave us just over 9 microseconds for the yield/context switch which constitutes the second part of our simulated interrupt. Because we have separately measured that value as 8 microseconds previously (in a loop with two processes constantly yielding to each other), we use 8 microseconds as our context switch time. We thus use 22 microseconds for the $t_{i,i}$. It is reasonable for it to be somewhat larger than the poll time, because even if the application is currently running, part of it should need to be saved to run an interrupt handler.

We used the time quantum value that Aegis uses, approximately 15 milliseconds, which is a standard time for several operating systems.

7.3.1 Ultrix-like parameters

In order to model a more Ultrix-like operating system running on the identical hardware, we modify some of the Exokernel parameters to create an aggressive, yet somewhat realistic, competitor. We assume that the ASH and upcall times stay the same. This assumption may be incorrect, it may take more overhead to implement ASHs and upcalls in a traditional operating system than it would on an Exokernel. The polling and interrupt time would certainly increase. We estimate those times to be equal to each other; the actual value we use is taken from Thekkath and Levy's measurement of the time to "Deliver Simple Exception to Null User Handler" [55] and scaled for our hardware (it was measured on a DECstation 5000/200). It is both an underestimate and an overestimate of what we believe the value to be: an underestimate because it obviously does not count the time spent in the AN2 device driver that we include in our measurements; an overestimate because it starts from the user application and therefore includes an extra bounce into the kernel. We assume that these two factors cancel out. We measured the context switch time under Ultrix (by comparing the roundtrip time to ping pong a word of data over UDP using non-blocking reads (polling) and blocking reads (interrupts). The measured difference was 87 microseconds (using blocking reads only on one side of the measurement); we chose 75 microseconds to be as fair as possible to Ultrix.

7.3.2 Workload parameters

We fixed two of the workload parameters across all of our studies. First, we assumed that there was only one other active process on the system (*i.e.*, $n_p = 1$). This only affects two of the calculations: polling when the process is suspended and blocked waiting for a message (blocked) and interrupt when the process is suspended and is not waiting for a message (busy-suspended).

We arbitrarily set f, the fraction of the handler that will run if the handler aborts to 1/8. Based on our application studies this is almost certainly an overestimate, that will hurt ASHs more than upcalls, and both of these more than user-level communication.

7.4 Verification

The model is intended to provide a rough idea of where each strategy should be used. We therefore attempt to provide a sanity check that the values coming out of the model are reasonable, by comparing the model to the experimental results seen in the remote increment experiment of Table 6-6, as shown in Figure 7-3. In order to more closely model this application, we change the model slightly: (1) we increase the sum of the interrupt time and the context switch time ($t_{i,i} + t_c$) from 30 microseconds to 45 microseconds to match the average time we measured and (2) we decrease the time to terminate a successful ASH and upcall ($t_{a,s}$ and $t_{u,s}$ as well as the address switch back) to zero, because they are overlapped with the transmission of the message to the remote side. For this application, the handler length was about 20 microseconds, and the sandboxing factor was 1.25. The probability of an ASH or upcall aborting was zero.

The left portion Table 7-6 shows the prediction of the model for these parameters. To compare these values to the application performance, we look at the predicted and measured differences between the communication models and the experimental data, in the right half of Table 7-6. As you can see, the numbers are in the right ballpark, but do not completely correspond. On the other hand, the trends correspond exactly: the relationship between any two pairs of model predicted numbers and measured data is the same.

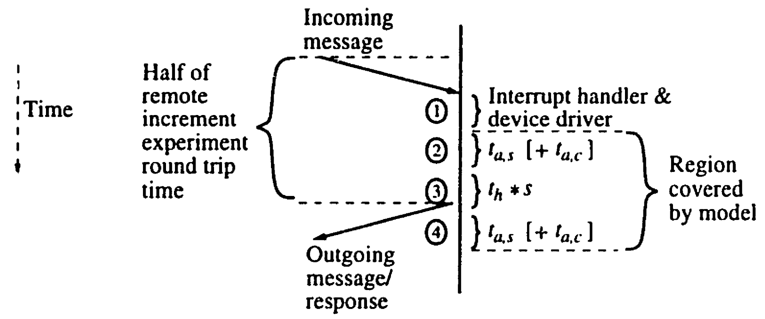

Time
Half of remote increment experiment round trip time
Incoming message
Outgoing message/ response
1
2
3
4
Interrupt handler & device driver
$t_{a,s}$ [+ $t_{a,c}$]
$t_h * s$
$t_{a,s}$ [+ $t_{a,c}$]
Region covered by model

Figure 7-3. This figure shows the steps that occur to perform the remote increment experiment used to verify the model, and shows which ones are modeled and which are not. To perform the verification, Step 4 was taken out of the model calculation, and the time measured for the remote increment was divided by two, and the measured message transmission time was subtracted from the experimental data.

Variable	Value
$t_{p,rp}$	40.88
$t_{a,rp}$	30.4
$t_{u,rp}$	46.18
$t_{i,sw}$	65.4
$t_{a,sw}$	30.8
$t_{u,sw}$	46.58

Quantity	Model prediction	Measured data
$t_{p,rp} - t_{a,rp}$	10.48	15
$t_{u,rp} - t_{p,rp}$	5.3	4.5
$t_{i,sw} - t_{a,sw}$	34.6	48
$t_{i,sw} - t_{u,sw}$	18.82	27
$t_{i,sw} - t_{p,rp}$	24.52	32.5

Table 7-6. The table on the left shows the predictions of the model for the remote increment handler described in Chapter 6. The table on the right shows how the predictions of the differences between the communication models correspond to the measured data.

Application	t_h	s	$p_{h,a}$
Remote increment	20	1.25	0
TCP latency	50	1.4–1.5	0
Web server (8-byte pages)	125	1.3	.1–1
Web server (2048-byte pages)	200	1.1	.1–1

Table 7-7. The application-dependent parameters that we measured for the experiments of Chapter 6. t_h is the handler run time (measured in microseconds), s is the sandboxing factor, and $p_{h,a}$ is the probability that a handler will abort.

7.5 Study

Given the model and the parameters, we can now use the model to make approximate predictions about the performance of applications. We will look at a range of handler lengths, representing the lengths of applications studied in this thesis.

For each set of parameters, we show four graphs, one representing the results for each process state (the process states were defined in Table 7-1). The x-axis of the graphs is the probability that the handler will abort: a value of zero means that it will never abort and one means that it will always abort. This parameter affects ASHs and upcalls. The y-axis of the graphs is the sandboxing factor. A factor of one means that the sandboxed handler takes the same amount of time to run as a non-sandboxed handler would; a factor of two means that the sandboxed handler takes twice as much time as a non-sandboxed one (we have seen the factor go as high as 1.6 in our experiments, but no higher). As shown in the legend, the gray level encodes which strategy yields the minimum time for a given set of parameters. Table 7-7 shows the values of the application-dependent parameters that we measured in Chapter 6.

We first look at a handler length of 20 microseconds. This corresponds to the length of the remote increment (ping-pong) handler measured in Table 6-6. The sandboxing factor was measured as 1.25 and the probability of abort for this experiment is 0. As seen in Figure 7-4, the model corresponds to the experimental data, as we would expect: whatever the process is doing at message arrival, using ASHs is the best strategy. Even with the highly conservative estimation we are using, ASHs outperform polling and interrupts over a very wide range of the space. As shown by Figure 7-5, the Ultrix-like parameters show ASHs to be useful over an even wider area of the space (since the overhead for polling and interrupts is higher). Upcalls are never faster for a handler this size, because the sandboxing overhead for ASHs is dominated by the overhead to initiate the upcalls.

Figures 7-6 and 7-7 demonstrate the expected performance if the handler run time is 50 microseconds. This corresponds to the handler length for the TCP latency experiment (ping-ponging four bytes). The sandboxing factor for this experiment was measured to be 1.4–1.5 (note that the more data sent per TCP packet, the lower the sandboxing factor) and the probability of abort was again zero. As shown experimentally (Table 6-7), polling outperforms ASHs at

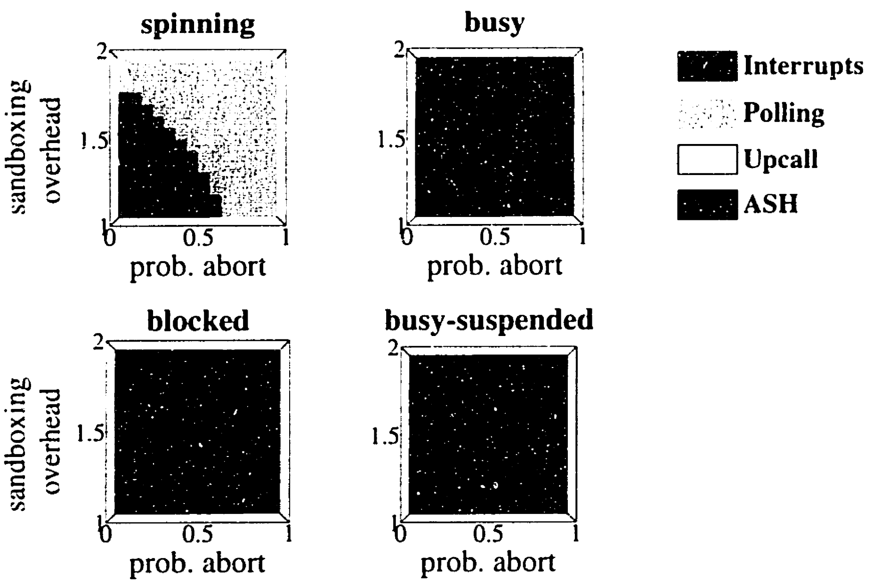


Figure 7-4. This graph models the behavior of the Exokernel running on the DECstation 5000/240 platform for a handler run time of 20 microseconds.

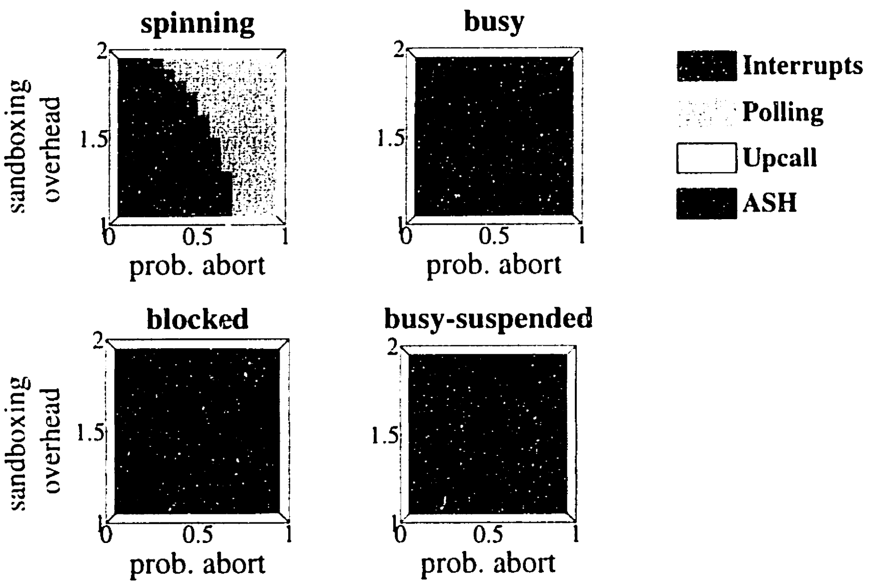


Figure 7-5. This graph models the behavior of Ultrix running on the DECstation 5000/240 platform for a handler run time of 20 microseconds.

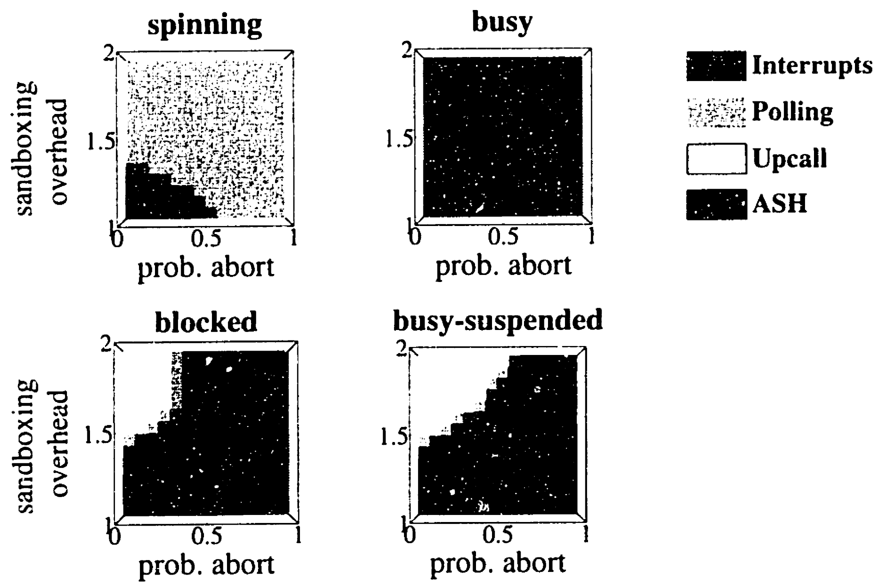


Figure 7-6. This graph models the behavior of the Exokernel running on the DECstation 5000/240 platform for a handler run time of 50 microseconds.

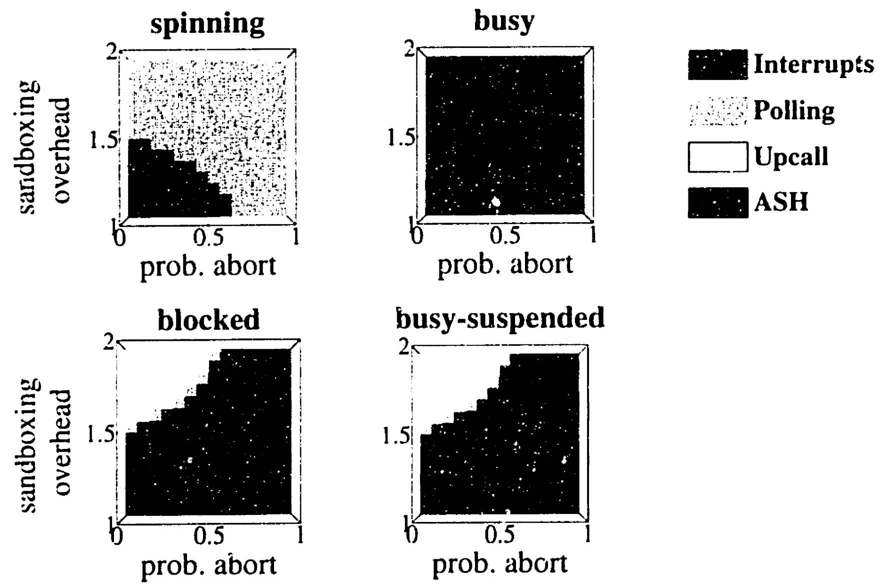


Figure 7-7. This graph models the behavior of Ultrix running on the DECstation 5000/240 platform for a handler run time of 50 microseconds.

this amount of sandboxing overhead when the process is scheduled. When it is not scheduled, the model predicts that the times should be similar; in fact, the interrupt version is a fair bit more expensive than ASHs, so again we have underestimated the interrupt penalty, and again we have slightly overestimated the upcall time.

With a greater handler length, the sandboxing overhead matters more when compared to the fixed initiation and termination costs. As we can see for the cases where the process is not scheduled under Ultrix, upcalls begin to be useful. As the sandboxing factor increases (along the y-axis), upcalls become cheaper than ASHs. However, as the probability of abort increases, the point where the crossover happens increases, because the overhead of initiating an upcall to see whether or not it will succeed is greater than that of initiating an ASH.

Figures 7-8 and 7-9 demonstrate the expected performance for the Exokernel and Ultrix if the handler run time is 125 microseconds. This corresponds to the web server application serving eight-byte pages. The experimental data agrees that ASHs will not do best at this handler length and with this sandboxing factor no matter what the probability of abort; the performance of interrupts is again over-predicted.

The main interesting point about these two graphs is that as the handler length increases, the range of where ASHs are useful decreases.

Finally, Figures 7-10 and 7-11 demonstrate the expected performance for the Exokernel and Ultrix if the handler run time is 200 microseconds. This corresponds to the web server application serving 2048-byte pages. All of the communication strategies both are predicted to have similar performance and show similar performance, except for interrupts, which again has worse-than-predicted performance.

Again, we can see the trend of the handler length increasing narrowing the range of ASH usefulness. The Ultrix-like operating system, however, still predicts ASHs and upcalls to be quite useful as long as the process is not scheduled.

7.6 Summary

In this chapter we developed a model to try and gain an understanding of the tradeoffs between using different communication models for different hardware, operating systems, and application workloads. Our model shows us several important trends. On non-Exokernel operating systems, both ASHs and upcalls are useful over a wide range of the parameter space. Because the Exokernel was designed for flexibility and performance from the start, it benefits less than other operating systems would from this functionality.

Sandboxing overhead can be a big factor in the choice between using ASHs and upcalls. As described in Section 6.2, we expect the factor to decrease with a better implementation of a sandboxer. Furthermore, other hardware may require a smaller sandboxing overhead (for example safety can be ensured inexpensively on the Intel x86 architecture through hardware support).

As shown by the model, when designing handlers, it is extremely important to consider the probability of abort for the application. Handlers which will abort often probably not worth

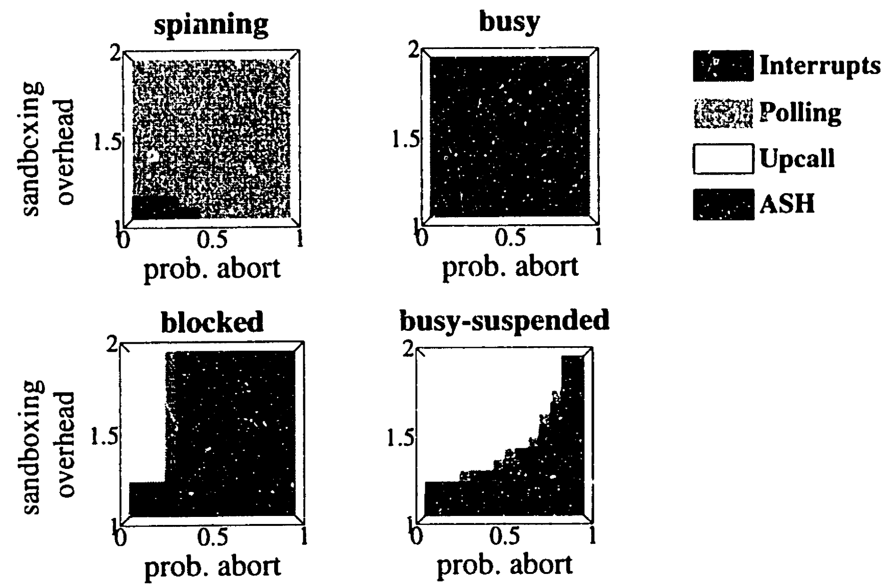


Figure 7-8. This graph models the behavior of the Exokernel running on the DECstation 5000/240 platform for a handler run time of 125 microseconds.

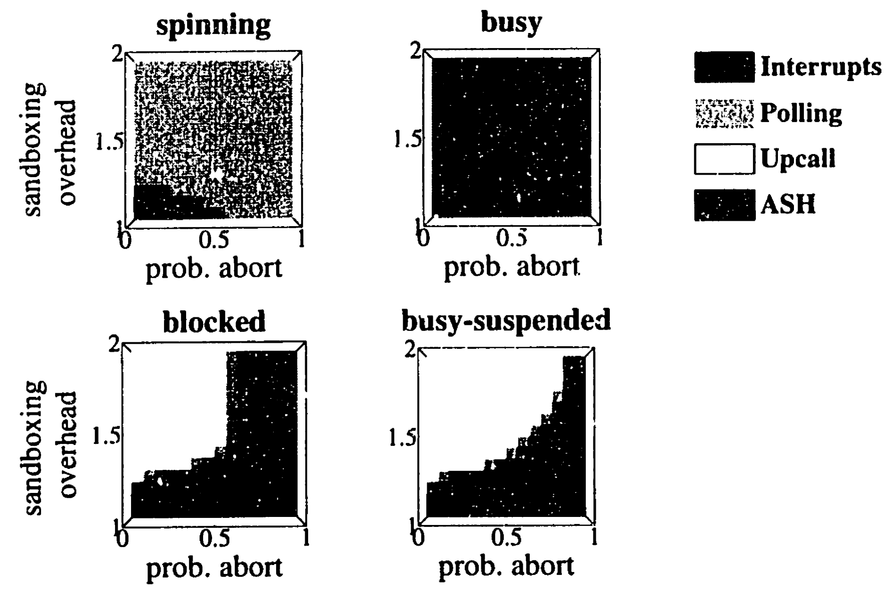


Figure 7-9. This graph models the behavior of Ultrix running on the DECstation 5000/240 platform for a handler run time of 125 microseconds.

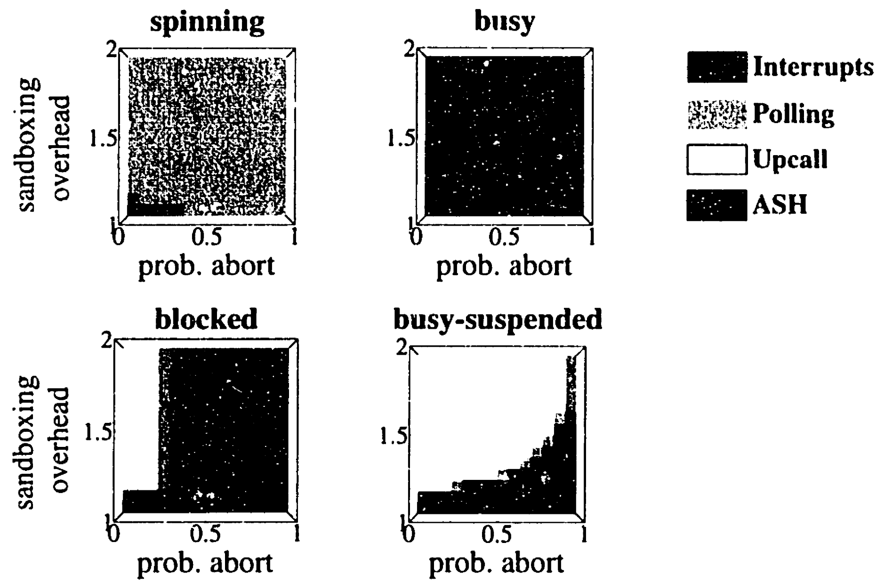


Figure 7-10. This graph models the behavior of the Exokernel running on the DECstation 5000/240 platform for a handler run time of 200 microseconds.

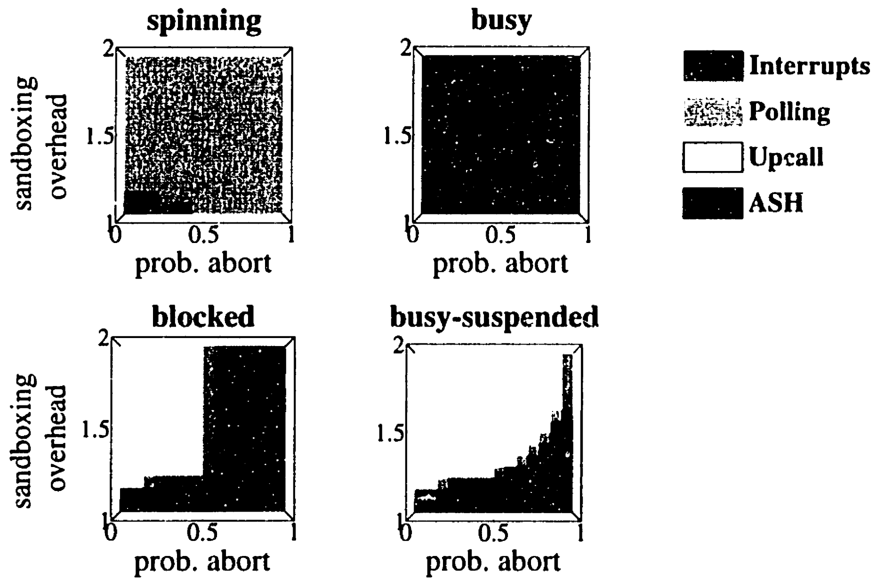


Figure 7-11. This graph models the behavior of Ultrix running on the DECstation 5000/240 platform for a handler run time of 200 microseconds.

making into handlers, since the cost of initiating and terminating them when they fail can be high. Handlers which are likely to succeed, however, appear to be useful in a variety of situations.

Chapter 8

Conclusion

This chapter is the final chapter of the thesis. The first section summarizes the results and conclusions of the thesis. The second section outlines directions for possible future work.

8.1 Summary

The goal of the work in this dissertation was to provide efficient communication performance to applications. Other work (*e.g.,* [31] and others) has shown that the flexibility of user-level communication can provide enormous performance gains to applications. Our work focused on enabling user-level communication to achieve high performance even in situations where it was previously unable to do so, by ensuring that incoming messages can be responded to both efficiently (in terms of system resources) and quickly (so as to maximize application performance).

This thesis presented the design and implementation of ASHs, a technique for delivering hardware-level network performance to applications by downloading application-specific code into the operating system. ASHs allow direct, dynamic message vectoring, message initiation, and control initiation, enabling a flexible response to incoming messages. We also presented the design and implementation of fast, asynchronous upcalls, an alternative approach to implementing handlers. Our design of upcalls is based on the ASH design, and can therefore achieve many of the benefits available to ASHs, especially (but not exclusively) the avoidance of the full context switch.

This thesis also provides a detailed description of the techniques necessary to run application code in the kernel safely even when the application programmer is not constrained to write in a pointer-safe language. Although the concept of sandboxing is not new [62], we extended its implementation by avoiding the exceptions ASHs could take, bounding their execution time, and limiting the available operating system interface in order to restrict the amount of OS changes that need to be made to support application code running in the kernel. We showed that the overhead of sandboxing can reduce the benefit of using ASHs, but are encouraged by the current research in the field, and expect that this cost can be greatly reduced.

This thesis also presented dynamic integrated layer processing (DILP). Through the use of DILP, data manipulations such as checksumming or conversions can be automatically integrated into the data transfer engine itself. Furthermore, because the DILP instruction loop is generated by the system for applications as opposed to by applications themselves, it results in a reduction of the sandboxing overhead that ASHs incur. DILP can be used with either ASHs or fast upcalls.

The thesis experimentally evaluated three different strategies to provide flexible communication to applications: unaugmented user-level communication, fast upcalls, and ASHs, using both microbenchmarks and end-to-end applications. We showed that ASHs and upcalls can provide high throughput, low-latency data transfer, and low-latency control transfer. We also demonstrated that ASHs and upcalls deliver performance consistently better than user-level communication alone when there are multiple processes. On the Aegis exokernel, the performance of ASHs and upcalls is similar, with ASHs out-performing upcalls for short handlers or for higher percentages of aborts.

ASHs have a lower cost of initiation and termination than upcalls do, and are more tightly coupled with the operating system. On the other hand, upcalls do not need to be sandboxed in order to be made safe. This thesis developed an analytical model of these three different strategies which allowed us to explore the tradeoffs as the architectural model, operating system, and application characteristics change. We saw that the benefits of ASHs and upcalls are highly underestimated by their performance on the Aegis exokernel, because the exokernel was designed with flexibility and performance in mind from the start. Furthermore, ASHs are predicted to outperform upcalls, even with the current sandboxing overheads, whenever the application cannot perfectly predict which handlers will have to abort.

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.

8.2 Future work

There are a number of further directions that can be explored starting with this thesis as a basis. Implementations of ASHs and upcalls on other platforms is one of the most basic extensions of this work, and would be extremely useful. In particular, the implementation of ASHs and upcalls on a more traditional operating system would expose any exokernel-dependent features we are assuming (if any). It would also allow us to actually experimentally verify the performance gains on a traditional operating system that we have currently only been able to model. Also illuminating would be an implementation on a different hardware platform. We are in the process of implementing ASHs on the x86 architecture, but it is as yet too early to conclude anything about the performance benefits.

Another direction which would be interesting is to improve the programming model available to ASH and upcall programmers. Currently, all pages must be pinned, and there are restrictions on the run time of handlers. If the kernel could suspend and restart a faulting or running-too-long ASH, the programming model would be simpler; we expect this would be a worthwhile tradeoff as long as it does not increase the initiation time of ASHs. Another potential improvement would be to allow ASHs to directly modify kernel data structures through the use of the Proof Carrying Code techniques of Necula and Lee [43], instead of using the system call interface.

A further exploration of dynamic protocol composition would be desirable. The problem with having the ability to specialize protocols and abstractions is that a great number of them can arise. If they can be written modularly and then efficiently combined, the job of the programmer is greatly simplified. A good implementation of dynamic protocol composition could thus be very useful.

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.A. Patterson, D.S. Roselli, 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] T.E. Anderson, S.S. Owicki, J.B. Saxe, and C.P. Thacker. High speed switch scheduling for local area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[4] 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.

[5] 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*, pages 267–284, Copper Mountain Resort, CO, USA, December 1995.

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

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

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

[9] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, USA, October 1996.

[10] J.B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis, Rice University, August 1993.

[11] 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.

[12] 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.

[13] D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, pages 200–208, Philadelphia, PA, USA, September 1990.

[14] 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.

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

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

[17] 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, Asheville, NC, USA, December 1993.

[18] 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*, pages 2–13, London, UK, August 1994.

[19] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, USA, October 1996.

[20] 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*, pages 14–24, London, UK, August 1994.

[21] 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*, pages 160–170, Philadelphia, PA, USA, May 1996. `http://www.pdos.lcs.mit.edu/~engler/ vcode.html`.

[22] 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)*, pages 53–59, Stanford, CA, USA, August 1996.

[23] 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, Colorado, December 1995.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] K.L. Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, Massachusetts Institute of Technology, December 1995.

[30] 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.

[31] M.F. Kaashoek, D.R. Engler, G.R. Ganger, and D.A. Wallach. Server operating systems. In *Seventh SIGOPS European Workshop: Systems Support for Worldwide Applications*, pages 141–148, Connemara, Ireland, September 1996.

[32] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–132, San Francisco, CA, USA, January 1994.

[33] 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.

[34] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. Early version appeared in Proceedings of SPAA '92, November 9, 1992.

[35] 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.

[36] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain Resort, CO, USA, December 1995.

[37] 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, Asheville, NC, USA, 1993

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

[39] J. Mogul. The case for persistent-connection HTTP. In *Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM '95)*, pages 299–313, August 1995. A more comprehensive version of this paper is available on line at Digitial Equipment Corporation Western Research Laboratory, Research Report 95/4 May, 1995.

[40] J.C. Mogul and K.K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. Technical Report 95/8, Digital Western Research Laboratory, December 1995. This report is an expanded version of a paper in the Proceedings of the 1996 USENIX Technical Conference.

[41] 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.

[42] 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.

[43] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, WA, USA, October 1996.

[44] V. Padmanabhan and J Mogul. Improving HTTP latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, USA, October 1994.

[45] 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.

[46] S.H. Rodrigues, T.E. Anderson, and D.E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 257–274, Anaheim, CA, USA, January 1997.

[47] D.J. Scales, M. Burrows, and C.A. Thekkath. Experience with parallel computing on the AN2 network. In *International Parallel Processing Symposium*, pages 94–103, Honolulu, HI, USA, April 1996.

[48] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, USA, October 1996.

[49] J.P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

[51] P.G. Sobalvarro. *Demand-based Cosheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, January 1997.

[52] R.W. Stevens. *TCP/IP illustrated: the protocols*, volume 1, chapter 18, page 237. Addison-Wesley Pub. Co., 1994.

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

[54] 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.

[55] C.A. Thekkath and H.M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 110–119, San Francisco, CA, USA, October 1994.

[56] 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 Francisco, California, October 1994.

[57] 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*, pages 64–73, San Francisco, CA, USA, October 1993.

[58] C. Tschudin. Flexible protocol stacks. In *Proc. SIGCOMM 1991*, pages 197–204, Zurich, Switzerland, September 1991.

[59] 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.

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

[61] 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 256–266, Gold Coast, Australia, May 1992.

[62] 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, Asheville, NC, USA, December 1993.

[63] 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.

[64] 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, California, August 1996.

[65] 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*, pages 217–226, Santa Barbara, CA, USA, July 1995.

[66] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[67] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, pages 153–165, San Francisco, CA, USA, January 1994.