Design and Implementation of a Modular, Flexible, and Fast System for Dynamic Protocol Composition

D. R. Engler, D. A. Wallach, and M. F. Kaashoek
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
{engler, kerr, kaashoek}@lcs.mit.edu

Abstract

Distributed systems must communicate. To communicate at all requires high-level protocols be built with manageable complexity. To communicate well requires protocols efficient both in design and implementation. The ASH system provides mechanisms to address both of these needs. To manage complexity, it provides a simple interface that allows protocols to be dynamically composed in a modular manner. As a result, the complexity of building a high-level messaging service is reduced since it can be built from multiple, independent pieces at runtime. To provide efficiency, the ASH interface is designed so that the message processing steps of each protocol (such as checksumming, byteswapping, encryption, etc.) can be dynamically integrated into a single specialized loop that touches each byte of the message at most once. The ASH system is the first to dynamically integrate the data processing elements of each protocol. This ability is crucial since without it dynamic protocol composition cannot be efficient.

This paper presents the design and implementation of the ASH system. It is interesting because it uses modern compilation techniques to eliminate the runtime overhead of modularity. To make the system efficient we have had to solve problems ranging from theoretical graph algorithms, to language design, to determining the most efficient way to access unaligned memory, and designing efficient and portable methods of generating executable code at runtime. As a result of careful design and the the use of dynamic code generation to specialize data processing paths, the ASH system provides performance that is comparable to (and in some situations significantly better than) hand-coded, hard-wired protocol implementations.

The ASH system has been integrated into the Aegis operating system, and runs on both Ethernet and DEC AN2 networks. A standalone version of the ASH system runs on SPARC, MIPS, and Alpha processors. The system itself will be publically released in the near future.

1 Introduction

Data is sent across a computer network using a network protocol. Protocols provide a more palatable interface than the raw network hardware (such as the provision of reliable messaging). Analogous to procedures in algol-derived languages, protocols are the units of modularity in networking. To provide higher level functionality than is provided by any single protocol they are frequently composed together into protocol stacks.

Protocol composition is used for three main reasons. First, it provides portability by encapsulating hardware dependencies in a low-level protocol, which can then be composed with higher level, medium-independent protocols. Second, it manages the high complexity of modern high-performance protocols by building up services from individual existing protocols via composition rather than reimplementation. This concern is not academic: implementation of most network protocols is complex and error-prone (e.g., the *specification* of TCP is more than eighty pages long [10]). Without composition the creation of high-level network protocols becomes an ambitious endeavor rather than a handful of function calls. Third, composition allows new protocols to be used with old networking software by encapsulating them in older protocols, thereby providing existing routers and transport software with packets that they can understand (and, more importantly, that they will not throw away as "garbage"). This use is important because network systems are distributed and, as a result, messages may travel through many different administrative domains, across many different network types and through many potentially outdated switching nodes. Without encapsulation, it would be difficult to communicate with other machines.

Traditionally, protocol composition has mainly been static in that compositions are determined at compilation time (the most famous static composition being the TCP/IP stack). While this approach works for simple cases, it has weaknesses in the face of demanding clients or of rich networking environments since many characteristics of the network (is it an Ethernet, a token ring, or ATM; is the link lossy, reliable, etc.) and of client needs (should throughput or latency be optimized for, what protocol services are needed or redundant?) are not known until runtime.

Since static composition leads to exponential code growth if every conceivable network protocol and, more compellingly, every protocol *implementation* is composed with every other, implementors have begun to defer composition until runtime. Runtime (or "dynamic") composition can be used to combine only those protocol stacks that are needed. Unsurprisingly, the flexibility of this approach has come with a cost. The most serious cost is that the different layers of the system are not integrated.

Additionally, integration across the protocol/application boundary has been poor in most systems. For example, hand integration typically combines only protocols (for example, a

This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

memory copy and checksum in TCP/IP) and does not integrate application-specific data manipulation operations, with the result that crossing the protocol/application boundary typically suffers multiple data traversals. Additionally, the model of application and network has been mainly a passive one: applications pre-bind buffers to store messages into; if the buffer to store the message into depends on the message contents, or the application has not installed enough bound buffers, the data is copied multiple times, obviously hurting performance.

We have developed a system that provides efficient protocol composition across both protocol and application boundaries and, due to its event model, allows applications to actively participate in message processing (both in determining where to place the message and in supplying operations to perform on it). The system has a simple but powerful interface that has been designed in tandem with a dynamic integration facility that dynamically compiles multiple data processing steps into a single specialized memory loop.

Since integration occurs at runtime, protocol integration does not require source compilation. Practical benefits of this fact are that: (1) protocols can be written in any language, (2) protocols can be compiled separately (or even not have any source code available on the system), and (3) protocols can even be dynamically linked as their implementations are upgraded.

One interesting problem we had to solve is that that current compilers do not consistently optimize networking idioms well. Therefore, while the protocols themselves can be written in any language, the data manipulation routines are written in a machine-independent language specialized for networking operations such as checksumming, bytewapping, memory copies, and unaligned memory accesses. As a result, we have been able to achieve performance beyond what is obtainable from current implementations of C.

This paper describes the design and implementation issues of the ASH system in detail. We are publishing a complementary paper on the use of ASHs [15].

Section 2 gives an overview of the ASH interface. Section 3 discusses challenges in the ASH implementation, and Section 4 gives details in how processing layers are dynamically integrated. The ASH operating system interface is described in Section 5 and experiments are presented in Section 6. Finally, related work is discussed in Section 7 and we conclude in Section 8.

2 System Overview

The ASH system provides support for dynamic, modular network protocol composition; after protocols have been composed, the ASH system uses on-the-fly code generation to integrate them into an efficient data path.

At a high-level, protocol composition consists of creating "protocol stacks". Each layer in the stack is linked to the layer immediately above and and the layer immediately below it. Oriented vertically, the low end of the stack is the link-layer protocol, and the high end the application (see Figure 1). When a message for a protocol stack is received, the ASH system delivers the message to the lowest protocol layer. Propagation from one layer in the stack to the next is terminated when the top of the stack is reached or when a layer decides that it has completely consumed the message. This upward path from link level to application level is called the *receive path*. The *send path* from application down to link level is similar, with the difference that messages originate from within the

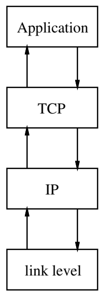

Figure 1: Protocol composition stack.

protocol stack itself and are propagated downward until either the link level is reached or until a lower level of the stack consumes them (e.g., as might happen on slow links where small messages are consolidated into larger ones).

There are three main actions a layer can perform on message data: *consume*, *augment*, and *deliver*. Logically, *consume* destructively copies portions of the message; this ability can be used to extract a protocol level's header(s). *augment* appends or prepends data to the message stream; this ability can be used to add headers or to reassemble fragments. Finally, *deliver* controls the actual movement of data between levels: it propagates a given number of bytes of the message to the next layer in the stack ("next" is determined by context: on sends it is the lower layer, on receives the upper). Since deliver and augment are always coupled, we have simplified the interface by unifying them into a single function.

The central efficiency problem with modular protocol design is that separation of protocol levels prevents integration of each protocol's data manipulations [3]. For example, consider a two-layer stack consisting of TCP and an RPC protocol that guarantees correct byte order. When a message is received and delivered to TCP, the TCP layer will touch the entire message to perform a checksum and then give the message to the RPC protocol which may then need to byteswap the entire message. Multiple message traversals are expensive given the current imbalance between memory and CPU speeds. The ASH system provides support to allow the message to be touched only once. Each operation takes an optional argument that specifies a *pipe* to apply during actual data manipulation. Using this mechanism, the data processing steps of each level can be integrated into a single pass over memory. As messages propagate between levels, the pipes to be applied to them (e.g., checksum, byteswap) are recorded; when the data is finally moved or touched, these steps are applied in a single pass over memory. To eliminate interpretation overhead, the code to perform this operation is dynamically generated at runtime, consolidating all operations into a single integrated routine, and cached on the path that created it.

This message processing interface is simple, powerful, and efficient. Simplicity results from careful design: we have reduced message processing to two main functions (consume and deliver) and augmented them with a simple mechanism to compose message processing steps. Fragmentation, reassembly, header addition and extraction can all be expressed using the same small interface. Power is derived from providing support for arbitrary message actions: supporting fragmentation, reassembly, adding headers to the beginning and end of the message, coalescing and splitting multiple messages into and out of a single incoming message, etc. Efficiency is derived from the careful application of dynamic code generation in order to integrate all message processing steps into a single specialized loop.

Table 1 shows the ASH interface in more detail. consume and deliver provide a rich yet simple set of potential message manipulations. For example, deliver can be used to append fragments to the beginning or end of the message in a manner identical to how it is used to append headers; similarly it can be used to deliver multiple application data units from a single message by simply using multiple invocations. initiate provides the capability for any layer to initiate a message send.

An example of how to use this interface to express a simplified version of UDP is provided in Figure 2. For clarity, our example only shows the receive path, and does not include the code necessary to handle the UDP pseudo-header. It consists of three parts: a main body, a commit handler, and an abort handler. The main body first initializes the accumulation register for the checksum to zero. It then uses consume to read the UDP header and remove it from the data stream, performing a checksum in the process. If nothing has gone wrong, it then checks the UDP header to find out if a UDP checksum was sent with the message. If none was sent, it uses deliver to forward up the rest of the data stream with no further data manipulations. If there was a checksum, it forwards up the rest of the data stream using deliver with a checksum pipe, then checks the checksum if the deliver completed successfully. If any problems happen at any time during the processing of the message, this simple ash aborts. The commit and abort handlers will be discussed later.

3 Ash Implementation Overview

This section examines four implementation challenges.

3.1 The travails of incomplete information

The overall efficiency goal of the ASH system is to process messages with minimal overhead. To process messages efficiently requires that messages be sent and received with a single copy between application data structures and network buffers (or, if message buffers can be mapped into user-space [5], with a single "touch" of the data). This goal presents a problem in that while messages are processed with only local information (i.e., incomplete knowledge of the entire message), many protocol decisions require global information.

This problem manifests itself on the receive path in that whole-message computations such as a checksum cannot be performed by low-level layers before propagating messages to higher levels. As a result, higher levels can consume a message before they are assured of message correctness. ASH uses *two-phase message processing* to handle this problem: phase one propagates the message up the protocol stack, phase two either commits or aborts. This model has similarities to that of Abbott and Peterson [1]; the main difference is that it is simpler.

As can be seen in Figure 2, the receive path of each protocol is comprised of three procedures: a body procedure that does the main processing during phase one, and two handlers, 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. The body procedure can fail, synchronously returning an error code to the level that invoked it via deliver; this level is then responsible for handling the failure (potentially by returning an error in turn to the level that invoked it). After the message is processed, abort or commit are invoked. During phase one, the ASH system tracks which handlers to call by recording the protocols invoked during message processing. Every time a protocol is activated its handlers are (logically) enqueued on a list. When phase two 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 path is affected by incomplete information as well. The problem is one of circularity: before a message header can be attached to the outgoing message it requires a checksum to be computed for the entire message, but in order to minimize copies this computation cannot be done until the message is moved to its network buffer destination, which cannot happen until the header is prepended, etc. To solve this problem we break the send path into two phases (similar to the receive path except that there is no abort handler) and introduce *reserved headers*. During deliver on the send path, protocol levels can reserve space for their headers by providing a special pointer value (defined by ASH_RESERVE) as an argument to deliver. deliver checks for this value and, if encountered, reserves the given number of bytes. When the message is consumed (either by the link layer or a lower level protocol) the send commits associated with each participating protocol are called in FIFO order. This ensures that layered checksums will work correctly, since these headers must be filled in from the highest layer to the lowest. Send commits must fill in any reserved headers that were allocated.

This methodology requires that data be fed to pipes out of order (since headers are applied after the main message). To determine if pipes can handle this behavior, each pipe is annotated with a flag indicating whether it is commutative: if any pipe to be applied is not commutative, the data is copied to a temporary buffer until the headers are ready, after that, the entire message is then copied, in order, to the final destination.

3.2 On-demand pipe generation

The number of possible pipe compositions grows exponentially with the number of pipes on a given path. This growth is further increased both since messages can originate and terminate at any level of the protocol stack and any given level can have an arbitrary number of possible pipes it is allowed to supply. This growth is a practical concern for protocol stacks of even modest size.

However, while the number of *potential* pipe compositions may be exponential, the number of *actual* compositions that are used are not. To exploit this fact, the ASH system integrates pipes on demand. The system starts with no integrated copy routines. When a set of composed pipes is to be applied, the ASH system checks to see if the corresponding integrated processing loop exists, if not it creates it. While this approach is space efficient, it also is time efficient since only those paths that are used are actually created dynamically.

```
/* This is the main body of the receive ash for UDP. */
/* nbytes is the size of the message. */
/* Note that the required checksum over the IP pseudoheader has */
/* been omitted for clarity.
int udp_receive_ash(void *ash_data, int nbytes) {
    struct udp udp_hdr;
    int sum, left;

    /* Initialize the checksum accumulation register which will be used in pipe. */
    p_export(CKSUM_PIPE, pipe_cksum, 0);

    if(nbytes<sizeof(struct udp)) /* Udp got too few bytes, abort now. */
        return(ASH_ABORT);

    /* Slurp the UDP header into a local variable (udp_hdr) */
    /* In the process, run the checksum pipe on the header. */
    /* If it fails, return abort. */
    if(consume(&udp_hdr, sizeof(struct udp), CKSUM_PIPE) < 0)
        /* Should not happen since we have enough bytes */
        return(ASH_ABORT);

    left = nbytes - sizeof(struct udp);

    /* If there is a checksum in the UDP header, */
    /* must finish the checksum computation at this point. */
    /* If the header checksum value is 0, no checksum was sent by the remote side. */
    if(udp_hdr.cksum != 0) {

        /* Deliver rest of message to higher level layer, using the checksum pipe. */
        if (deliver(left, CKSUM_PIPE) < 0) return(ASH_ABORT);

        /* Read the checksum accumulation register to find out the final result */
        /* of the checksum now that the deliver has finished. */
        sum = p_import(CKSUM_PIPE, pipe_cksum);

        /*  Fold 32-bit sum to 16 bits and invert (part of the checksum computation). */
        sum = (unsigned short)~inet_fold(sum);

        if(sum) /* Should have summed to zero.  Checksum failed. Abort.  */
            return(ASH_ABORT);
    }
    else {  /* No checksum to check */

        /* Deliver rest of message with no pipe. */
        if (deliver(left, ASH_SPECIAL_PIPE)<0) return(ASH_ABORT);
    }

    /* Ash succeeded! */
    return(ASH_SUCCESS);
}

/* This is the UDP receive commit handler.  */
int udp_rcommit(void *ash_data) {  /* Pass on success */
    return(ASH_SUCCESS);
}

/* This is the UDP receive abort handler.  */
void udp_rabort(void *ash_data) {  /* No cleanup to do; pass on failure. */
    return(ASH_ABORT);
}
```

Figure 2: Example ASH for UDP.

Operation	Description
int consume(void *dst, int nbytes, int pipe_id)	Consume nbytes of message by copying it to dst while applying pipe pipe_id. Used to consume headers, fragments, and application data. Derivatives of this function are provided to copy or skip over data.
int deliver([void *pre_hdr, unsigned pre_bytes, pipe_id pre_id,] unsigned nbytes, pipe_id pipe_id [, void *post_hdr, unsigned post_bytes, pipe_id post_id])	Deliver nbytes of the incoming byte stream to the next protocol (on send, this is the protocol immediately below the current protocol; on receive it is the protocol above). The pipe pipe_id will be applied when the data is touched. Optional arguments append data to the front (pre_hdr) or back (post_hdr) of the message; each of these pieces of data can be manipulated by their own pipes. deliver causes the protocol's final code to be pushed. (deliver also has a more complex form that takes a list of data elements to append to the message.)
int initiate(void *data, int nbytes)	Initiate a message send of nbytes of data. Causes the current protocol's send path to be invoked.

Table 1: ASH Interface

3.3 Fast pipe path computation

Protocol levels are allowed to select from among any number of different pipes to apply to a message. The problem this raises is efficiently determining what sequence of pipes should be applied when a message is finally consumed. A simple (but slow) solution would, for each piece of data, accumulate a list of pipes applied to it. When the message is consumed, this list would be used to derive the required copy routine. Such a methodology is far too inefficient. Ideally we would like to record the path we took through the protocol stack with a single integer operation (e.g., add) that maps to a dense set of integers corresponding to each possible set of pipes (density is required so we can use the path id as a table index).

This problem is reducible to a fast graph algorithm by considering each protocol level as a node and the pipes of a level as edges connecting one node to another. The problem we want to solve can be stated as follows:

> Given a leveled graph, label its edges so that any path starting at any node and ending at any node that passes through each level at most once can be labeled with a unique identifier in the range $0..n-1$ where n is the number of possible paths through the graph.

Such a path corresponds to the path a message takes through the protocol stack. The following algorithm gives a labeling of the stack that can be used to accumulate the path that we took from one level to another by simply adding the unique pipe id for a given path:

```
% Given a protocol stack where each level of the
% stack has a list of pipes associated with it
% numbered 0 .. m, create a new labeling s.t. the
% addition of paths from any two nodes that pass
% through each protocol node is unique and dense.
% Note that the identity function is pipe_id 0.
procedure label_pipes(protocol)
    % max holds the maximum sum we have seen
    max = 0;
    % traverse the protocol stack bottom up
    for protocol = bottom to top do
        pipelist = protocol–>pipelist;
        for i = 0 to pipelist–>size do
            pipelist[i].pipe_id = i ∗ (max + 1);
        % the new max
        max = max + pipelist[i].pipe_id;
```

This sum will serve as a dense index into a table containing specialized copy routines. If the routine we want is not there, the path can be re-derived in a simple, efficient manner by working backwards from highest numbered level to greatest as follows:

```
% Given a path summation, reconstruct the path
% traversed to get to it. The set of
% pipes along this path is put on a list and then
% dynamically compiled.
procedure recover_path(protocol, path_id)
    pstack = ();        % will hold the list of pipes to apply

    % start from the highest numbered level.
    for protocol = top to bottom do
        pipelist = protocol–>pipelist;

        % foreach p = largest pipe to smallest
        for i = pipelist–>size to 0 do
            if pipelist[i].path_id ≤ path_id
                push(pstack, pipelist[i]);
                    path_id = path_id – pipelist[i].path_id;
                    continue;
     % return the stack of pipes.
     return pstack;
```

A key property of this algorithm is that traveling from one level to another requires only a single integer addition. Importantly, the only message buffer modifications we must do are when the message buffer is created: since all buffers pass through the same pipe (as they are sent from one level to the next) their initial value can be added to the sum accumulated as the message traverses the protocol stack. When the message is to be consumed, we simply add the final sum to its initial sum (the initial sum is the pipe id of the level that it was first created with minus the path sum computed up to that level).

3.4 Interface tricks

The ASH interface has been designed to allow simple but powerful message manipulations to be done without copying overhead. Two of the useful interface features provided are that fragmentation and reassembly can be performed using the same interface and actions used for more "normal" operations (such as appending and extracting headers). Fragmentation is implemented by consuming a given fragment and storing it in a temporary message buffer; reassembly is simply a matter of glueing this buffer to any other portions of the fragment during a deliver to a higher-level. Similar techniques can be used to coalesce multiple messages into a single large message.

Another example of simplicity that can be used to provide powerful features is the ability to produce multiple higher-level messages from a single low level message. This ability means that multiple outstanding receive operations can be active at a single time. To support this, we have wrapped the global state each path requires in a context that can be cheaply switched on each activation.

Messages can be initiated by any ASH level. This ability allows low-level layers to send acknowledgments and retransmissions as necessary. Similarly, any layer can absorb messages sent by higher layers in order to buffer them, etc.

4 Dynamic ILP Implementation

The main technical contribution is the support the ASH system provides for dynamic integrated layer processing. This section looks at the issues we have had to solve in order to make dynamic ILP work well.

4.1 A digression on dynamic code generation

Dynamic code generation is the generation of executable code at runtime. While becoming a widely cited curiosity, it is still used by only a minority of implementors. To give the reader background in the important intuitions behind it, we present two of the uses we have put it to: (1) to eliminate interpretation overhead and (2) to reconcile modularity with performance by compiling away indirection.

Dynamic code generation is compilation. Like static compilation, dynamic compilation can eliminate interpretation. Interpretation has been given fairly heavy handed semantics; in this context it simply means that data structure values determine control flow.

Typically, a data structure changes slowly in comparison to how many times it is used (initialize a matrix once, then perform many passes over it). Furthermore, by definition, the control path the data structure determines could be specialized to the given data.

These are simple observations. However, their applicability is extraordinarily wide. Every non-trivial program interprets. If specialization can be performed quickly, and insight into how it can be applied can be found, every existing program could be made to go faster.

Systems programming is especially amenable to dynamic code generation: systems functions are commonly used by many different application classes (e.g., printf is called by almost every C program). To meet the demands of different applications, systems software is very general. What generality means in this context is that it uses much interpretation.

Traditionally, modularity has been expensive: compile-time or runtime boundaries have been expensive to cross both in terms of the number of cycles required to implement the crossing and in the truncation of information that they forced (and subsequent elimination of possible optimizations).

The use of dynamic code generation allows modularity to be eliminated by compiling away hard boundaries, thereby tightly integrating the formerly separated pieces of code. This ability both improves performance dramatically and, perhaps more importantly, provides a mechanism with which complex systems can be constructed without fear of expense.

4.2 Pipe overview

To make the ASH system efficient we have had to design efficient and portable methods of generating executable code at runtime.

As in Abbott and Peterson [1], each protocol layer negotiation step for data manipulation is written as a transfer from register to register, and is organized as a "pipe" manipulating data as it streams through. This methodology is similar to Abbott and Peterson [1] with the important distinction that our system allows dynamic protocol composition, whereas theirs is done statically. 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 the checksum's output to be streamed through the encryption pipe, it is aggregated into a single register. The composed functions are compiled by our system into a tightly integrated streaming engine which is encoded into a specialized data copying loop.

The pipes for ASHs are written in a variant of VCODE [6], which is a low-level extension language designed to be simple to implement and efficient both in terms of the cost of code generation and in terms of the computational performance of its generated code. The VCODE system was built based on our early experience with the ASH system.

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 is provided in Figure 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 the pipe during use. 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 (P_MOD), 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., can it be reordered, and the expected input and output sizes) and how it can be used (e.g., can it process message headers after message data in order to compute message checksums).

Since pipe operations are written in terms of portable assembly language instructions, pipes are charged with allocating those registers they need and are given control over register class. The two register classes are *temporary* and *persistent*. Temporary registers are scratch registers that are not saved across pipe invocations. Persistent registers are saved across pipe invocations; they are used, for example, as accumulators during checksum computations. The values of persistent registers can be imported and exported from the main protocol code. Export is used to initialize a register before use, 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. 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 (using the p_cksum32 instruction), and then outputs it. Since the pipe does not modify its input we associate it with the P_NO_MOD attribute. The pipe compiler

```
/* Create a pipe to compute the Internet checksum; insert it on the pipe list. */
static void mk_cksum_pipe(int pipe_id, struct pipel *pl, reg_t *pipe_cksum) {
	/*
	 * Start allocating a pipe: inet checksum works with 16 bits and is both commutative
	 * and non–modifying (i.e., it does not alter its input).
	 */
	pipe_lambda(pl, pipe_id, P_GAUGE32, P_COMMUTATIVE | P_NO_MOD);
		/* Allocate an accumulator register (preserved across pipe invocations. */
		*pipe_cksum = p_getreg(pl, pipe_id, P_VAR);

		p_input32(p_inputr);  /* Get input word */

		p_cksum32(*pipe_cksum, p_inputr);  /* Add input word to running checksum computation. */

		p_output32(p_inputr); /* Output word */
	pipe_end();     /* done. */
}
```

Figure 3: Example pipe.

can use this information to reorder pipes as necessary to reduce the number of gauge conversions required. Additionally, since the Internet checksum computation is commutative, it can be applied to the data in any order. To indicate this attribute we use the P_COMMUTATIVE flag.

4.3 Language issues

Current compilers do not optimize networking idioms well. This is mainly due to the fact that there are no clear idiomatic way of expressing common networking operations such as checksumming, bytewapping, memory copies, and unaligned memory accesses from within even a low-level language such as C. To remedy this situation, we have extended the VCODE system (to which the pipe language is compiled to) to include common networking operations. Importantly, the VCODE system on which ASH is built is low-level enough that further extensions can be done by clients of the ASH system with little performance impact.

Clear expression of common data manipulation operations allows us to profitably tune the ASH system to exploit architecture-specific features. For example, the x86 machine has a byteswap instruction, and the Alpha has a rich set of byte manipulation routines. Unaligned access instructions are provided on the MIPS, ALPHA and x86 (useful for copying unaligned message data) and add-with-carry instructions are provided on the SPARC and x86 architectures (these instructions can be used to efficiently compute the Internet checksum a word at a time). Isolating machine-specific details to the ASH and VCODE systems allow clients to reap the benefits of aggressive specialization while retaining both portability and ease of use.

A practical result of such specialization is that we have been able to achieve performance beyond what is obtainable from current implementations of systems programming languages such as C.

A crucial aspect of the VCODE system is that it gives access to the underlying hardware at a lower level than a systems language such as C, while still maintaining portability. As described above, we have used this control to provide an extended set of network specific idioms. This control also has a number of other advantages. An example is the register spill problem that plagues static ILP systems due to their use of high-level languages to implement pipes: if pipes are integrated together recklessly and the variables they use exceed the number of registers of the underlying machine, ILP can actually be substantially slower than non-ILP code since each loop iteration will involve spilling and restoring some number of registers. VCODE was designed to solve precisely this sort of problem: VCODE provides a portable way to determine when the machine's registers are exhausted. At this point, the ASH system is able to split the copy operation into multiple passes.

4.4 Implementation mechanism

Internally each pipe is represented by a collection of persistent and temporary register names, an index into a global register table (used to translate the local register names into a global per-message register context) and an array of pipe instructions. Given a list of pipes to integrate, we perform the following operations:

1. Traverse all pipes, accumulating both the maximum number of temporary registers needed by any pipe, and the total number of persistent registers needed by all pipes.

2. Allocate the required temporary and persistent registers. If the underlying machine runs out of registers, split the loop into multiple passes.

3. Load the values of all persistent registers held in the per-message context into their allocated registers.

4. Generate an unrolled memory access loop: pass each loaded value from one pipe to the next. If the gauges between the pipes differ, perform any necessary conversions.

5. Store the persistent registers into the per-message context.

As described above, pipe compilation is built on top of the VCODE system. We exploit VCODE's low-level interface to craft specialized memory copy loops that, in many cases, can exceed the performance obtainable from high-level languages. Three examples of how pipe compilation exploits VCODE's low level interface are: (1) use of unaligned memory operations to make unaligned copies efficient, (2) determination of whether memory loads are done using an index register allowing message indexing to be implemented efficiently, and (3) exploiting machine support for "double loads" to load two registers at a time. The high degree of control VCODE gives (while maintaining portability!) means, again, that the quality of pipe-generated code can surpass that possible from a high-level language. Finally, on machines for which it is profitable, the pipe system can violate VCODE abstractions to generate customized architecture-specific copy loops. Importantly, the memory copy generator's complexity is isolated to a few hundred lines of code in the pipe compiler, but is leveraged by all clients of the ASH system, eliminating the tedious and error prone process of repeatedly implementing unrolled memory operation loops. As the number of steps to integrate increases, this benefit is non-trivial.

5 Integration With Operating System

After a message arrives, it it passed into the ASH associated with the message, if one exists. If the ASH aborts, the message is saved and passed on to higher levels later.

The message send function takes a pointer to be set when the message buffers sent along that path have all been freed and the ASH can reclaim the storage. When a message is going to be sent from an ASH to a network device, the device is called with the number of bytes to be delivered, and how many segments the bytes are in. The device can can either return a pointer to a buffer to copy the message into, or nil indicating that it can be left in place. When the message is ready, the device will then be called again, this time with the message. A network device such as the Ethernet, which can only send a message from specific network buffers, would always provide a buffer at the reservation time. On the other hand, a device such as the DEC AN2, which can gather-DMA a message with great flexibility, would probably leave the data inplace. One example where it might not leave the message in place was if it were very small but segmented into many buffers, since the cost of gather DMA-ing is greater than the cost of copying small messages.

6 Experiments

In this section, we examine how ASHs can be exploited to achieve good throughput and data transfer latency. Many of our experiments are influenced by Clark and Tennenhouse [3]. We use micro-benchmarks to measure the base performance of individual ASH operations. This methodology allows us to isolate the ASH system from artifacts of specific architectural and OS implementations.

6.1 Data manipulation costs

We perform experiments to show that data manipulation using pipes is as fast or faster than hand-integrated data routines.

Table 2 shows that the integrated routines generated by the ASH system are as fast or even slightly faster than that generated by gcc for the equivalent manipulations hand-integrated in C. We measured the time to repeatedly perform a combined byteswap with memcpy, checksum with memcpy, and byteswap checksum with memcpy on 4096 bytes of data, both with and without pipes, on a Decstation 5000/200. We were careful to make sure that the hand-integrated and the pipe-generated routines are unrolled by the same amount (4 words).

manipulation	hand-integrated	ASH
byteswap	8.07	8.42
checksum	15.8	17.2
checksum+byteswap	6.51	6.71

Table 2: Throughput measured on the Decstation 5000/200. Units are megabytes/second.

manipulation	hand-integrated	ASH
byteswap	19.2	23.5
checksum	30.1	63.5
checksum+byteswap	16.0	20.7

Table 3: Throughput measured on the SPARCstation 5. Units are megabytes/second.

Note that the cost of generating a pipe was under a millisecond. We expect this time to decrease significantly as our system gains maturity.

Table 3 shows the same experiments as Table 2, only for a SPARCstation 5. This experiment demonstrates that automatically integrated pipes using VCODE can provide substantial performance benefits over hand-integrating functions. There are two main reasons for the the dramatic superiority of the ASH system integration over the hand-crafted loops; both reasons result from the close ties to the hardware that the ASH system can exploit. First, the SPARC provides memory instructions to move two integer registers at a time to and from memory. The C compiler we compared against did not exploit this ability and hence performed twice as many memory operations per iteration. The second performance boost was the fact that the SPARC provides a carry bit that can be used to cheaply compute the Internet checksum 32-bits at a time. The C version of this code had no portable way to access this bit and so was forced to use boolean expressions to detect overflow. The combination of these two features allows the ASH to achieve a throughput rate of more than double that of the hand-integrated C code. Similar benefits are accrued whenever there are conflicts between the expressiveness of high-level languages and the capabilities of the underlying hardware. It is important to emphasize that the ASH system exploits such features invisibly to the programmer, who merely sees data manipulations as relatively straight-forward (and *portable*) pipe compositions.

6.2 A multiple layer ASH

We perform a simple experiment to determine the overhead added by dynamic integrated layer processing with the ASH system. We built a three-layer protocol stack, consisting of the IP protocol, the UDP protocol, and a "ping" application. The IP and UDP layers are the standard Internet protocols, and between them involve one checksum over the data per send or receive[1]. The application receives the data sent to it into a buffer of its own choice, decrements a four-byte counter in the first four bytes, and sends a new message directly out of that buffer back down through the UDP and IP layers. We call this experiment `ping`. Optionally, the application byteswaps the entire message both while receiving and sending it; this version of the experiment is called `ping-bswap`. We wrote

[1] We perform the full Internet IP and UDP protocols except for IP fragmentation and reassembly.

experiment	nbytes	non-ASH	ASH
`ping`	4	22.2	94.6
`ping`	4096	876	587
`ping-bswap`	4096	1630	1360

Table 4: ASH system costs measured on the Decstation 5000/200. Units are in pings/microsecond (each ping is the time to completely traverse up the three levels to receive the message then back down again to send the next one).

two versions of these experiments: one using ASHs and one without. The one without does no integration of the checksums, memory copies and (if necessary) byte swaps.

The dynamically integrated `ping` and `ping-bswap` applications are not possible in any other existing system. The price paid for this integration is modest: as shown in the first line of Table 4, the time to perform the `ping` experiment with only four bytes of data (i.e. just the counter) is only 70 microseconds more than the version that does not use ASHs. We believe most of this difference to be constant overhead, primarily due to the fact that all namespaces in the ASH system are local rather than global (e.g., the names of pipes are local to a given level, as are the names of its registers rather than residing in a global name space). As a result every ASH manipulation involves a fair number of translations in order to map local names to global ones. We are changing this portion of the ASH system and expect to reduce its overhead substantially. Other portions of the cost are due to structuring rather than the ASH system itself. For example, the non-ASH version of IP and UDP can compute the pseudo-header checksum more efficiently than the ASH version since it can directly manipulate the message.

As shown in the bottom two lines of Table 4, when we perform the `ping` or the `ping-bswap` experiment with actual work to do (4096 bytes of data instead of four), the base overhead of using the ASH system is completely hidden. In fact the ASH system is able to exploit integrated layer processing to combine the data manipulation steps overhead, and is therefore significantly faster than the non-ASH version. Even in our not-yet-highly-tuned system, we can provide the benefit of integration while still allowing modularity.

6.3 Experience with ASHs

The ASH system described in this paper is the third revision of a system developed over the course of the fifteen months. The first version of the system was developed to download message handlers into an operating system kernel where they could efficiently direct message placement (eliminating intermediate copies), perform integrated operations on the message as it was being placed in memory, and even initiate messages from the interrupt handler [7]. We are in the process of reintegrating the current ASH system into the Exokernel operating system on which the original was built. As the following experiment shows (taken from [7]) the use of ASH's allowed us to achieve performance beyond that of even hard-coded in-kernel message routines.

Table 5 shows the roundtrip latency over Ethernet of ASH-based network messaging and compares it to ExOS without ASHs, Ultrix, and FRPC [12] (the fastest RPC in the literature on comparable hardware). Roundtrip latency for Aegis and Ultrix was measured by ping-ponging a counter in a 60-byte UDP/IP packet 4096 times between two processes in user-space on DECstation5000/125s. The FRPC numbers are taken from the literature [12]. They were measured on a DECstation5000/200, which is approximately 1.2 times faster than a DECstation5000/125 on SPECint92.

Machine	OS	Roundtrip latency
DEC5000/125	ExOS/ASH	259
DEC5000/125	ExOS	320
DEC5000/125	Ultrix	3400
DEC5000/200	Ultrix/FRPC	340

Table 5: Roundtrip latency of a 60-byte packet over Ethernet using ExOS with ASHs, ExOS without ASHs, Ultrix, and FRPC; times are in microseconds.

The message processing at each node consisted of reading the 60-byte message, incrementing the counter, copying the new value and a precomputed message header into a transmission buffer, and then sending the reply message. In comparison to a complete application-level implementation, ASHs save 61 microseconds.

Despite being measured on a slower machine, ExOS/ASH is 81 microseconds faster than a high-performance implementation of RPC for Ultrix (FRPC) running on DECstation5000/200s and using a specialized transport protocol [12]. In fact, ExOS/ASH is only 6 microseconds slower than the lower bound for cross-machine communication on Ethernet, measured on DECstation5000/200s [12].

A number of more recent experiments measured on top of the Exokernel system will appear in [15].

7 Related work

There have been many instances of ad hoc ILP, for example, in many networking kernels [4]. There is also quite a bit of work on protocol composition [8, 14, 9, 2, 13].

The 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: (1) their system is intended for static composition, whereas our system allows dynamic composition, and (2) 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 [11].

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.

8 Conclusion

The ASH system provides a framework to build network protocols efficiently and well. It is the first system to provide dynamic integration of layer processing steps. Without this ability, previous systems have had to resort to copying messages multiple times as the message is moved across modularity boundaries.

A surprising result of this paper is that modular, automatic layer integration can be significantly more efficient than even non-modular, hand-crafted code. For example, an implementation of the internet checksum automatically integrated into a memory copy loop by the ASH system was over a factor of two more efficient than a hand-crafted loop on the SPARC architecture. This result is mainly due to two factors. First, we have designed a language that provides portable access to key network operations not usually available to programmers. Implementors can use this control to implement many network operations more efficiently than is possible using current compilers. Second, we have aggressively exploited dynamic code generation to eliminate interpretation.

While the system provides efficient support for message processing, its interface is both simple and powerful. A restricted number of primitives can be used in regular ways to perform a rich set of actions.

An important practical benefit of our approach is that it does not require source code in order to integrate layers. As a result, it can be used to couple routines available only in system libraries or even to routines that are dynamically linked in. The most important benefit of this methodology is that the application can be integrated into message processing in the same way as protocols themselves. Such integration is crucial in order to eliminate multiple data manipulations that would otherwise result from crossing the application/protocol boundary.

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] N.T. Bhatti and R.D. Schlichting. A system for constructing configurable high-level protocols. In *Proc. SIGCOMM 1995*, pages 138–150, August 1995.

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

[4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6), 1989.

[5] 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, October 1994.

[6] 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*, May 1996.

[7] 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*, December 1995.

[8] Robert Harper and Peter 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.

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

[10] J. Postel. Transmission control protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[11] T.A. Proebsting and S.A. Watterson. Filter fusion. In *Proc. POPL*, January 1996.

[12] 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.

[13] C. Tschudin. Flexible protocol stacks. In *Proc. SIGCOMM 1991*, pages 197–204, August 1991.

[14] R. van Renesse, K.P. Birman, Roy Friedman, Mark Hayden, and David Karr. A framework for protocol composition in horus. In *Proc. Principles of Distributed Computing*, pages 138–150, August 1995.

[15] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96) (to appear)*, Stanford, California, August 1996.