Automated Integration of Communication Protocol Layers

Mark B. Abbott and Larry L. Peterson[1]

TR 92-24

Abstract

Integrating protocol data manipulations is a strategy to increase the throughput of network protocols. This paper presents a program structure that can be used to integrate a broad range of protocols. This program structure provides a basis for the automatic synthesis of integrated protocols, and serves as a guide to custom, hand-coded implementations. In addition to describing this program structure and showing how integrated protocols can be synthesized, the paper also examines the performance limits and scalability of strictly layered versus integrated protocol implementations.

December 31, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This research was sponsored in part by DARPA Contract DABT63-91-C-0030.

1 Introduction

Data manipulation—e.g., encryption, presentation formatting, compression, computing checksums—is one of the costliest aspects of data transfer [3, 4, 6]. This is because reading, and possibly writing, each byte of data in a message involves memory loads or stores, which are relatively slow operations on modern RISC architectures.

Clark and Tennenhouse [4] suggest a strategy, called *Integrated Layer Processing* (ILP), for addressing this problem.[1] In this paper, we refer to ILP as *protocol integration*, or simply *integration*. The aim of integration is to improve throughput by reducing the number of loads and stores of message data performed by protocols. Normally message data is passed from one protocol to the next in the form of a complete message, buffered in a system data structure. Any protocol that manipulates the data in the message must load and possibly store each byte of the message. Integration combines data manipulations from a series of protocols into a pipeline that shares accesses to the message data structure, thereby reducing the number of memory accesses. Another way to say this is that the integrated implementation *streams* message data between protocols using registers.

To date, integrated implementations have been realized in only a few special cases, using ad hoc solutions. One contribution of this paper is a specific program structure for integrated protocols that applies to a broad range of protocols. One advantage of this structure is that it lends itself to the automatic integration of protocols. A second contribution is a study of the performance limits and scalability of integrated versus strictly layered protocol implementations. The rest of this section motivates integration from a performance perspective, and justifies a structured approach to implementing integration.

1.1 Motivation for Integration

The load and store operations required by data manipulations are expensive, typically ranging from 8 to 32 clock cycles per memory access [8], in contrast to other operations that complete in a single cycle on modern RISC architectures. Furthermore, the discrepancy between processor and memory performance is expected to get worse.

Caches offer only a partial solution to this problem. There is still a cost in accessing the cache—typically 1 to 4 clock cycles [8]. This cost must be paid by every data manipulation protocol, for every word of data. In addition, there are delay slots following each read access which may not all be fillable. The number of delay slots tends to increase as processor speeds increase, and superscalar architectures effectively increase the number of delay slots by making it possible to execute multiple instructions in a single cycle. A second shortcoming of caches is that while they are very effective in reducing memory accesses for many computations, the characteristics of strictly layered message processing are such that caching is not as effective [6]. Cache effectiveness depends on locality of reference, and layered message processing exhibits

[1] ILP also refers to the *design principle* that protocol specifications should not preclude the application of the ILP *implementation strategy*.

poor locality.

Protocol integration addresses this problem. Integration generalizes the compiler optimization known as *loop fusion*, illustrated in Figure 1. The for-loops in Figure 1(a) are like a strictly layered, serial implementation of data manipulations, and the for-loop in Figure 1(b) is like an integrated implementation of two data manipulations.

```
for( i = 0; i < 10000; i++ )
      msgData[i]++;                  /* LOAD, ADD, STORE */

for( i = 0; i < 10000; i++ )
      msgData[i] = ~msgData[i];      /* LOAD, COMPLEMENT, STORE */
```

(a) Two For−Loops

```
for( i = 0; i < 10000; i++ ){
      temp = msgData[i];             /* LOAD */
      temp++;                        /* ADD */
      temp = ~temp;                  /* COMPLEMENT */
      msgData[i] = temp;             /* STORE */
}
```

(b) Integrated For−Loops

Figure 1: For-Loops

The motivation for loop fusion is to reduce loop overhead by sharing a single loop, and to expose the combined loop body to additional optimizations. In contrast, the primary motivation for integration of protocol data manipulations is to expose the combined loop body to one particular optimization: to pass each unit of data from one data manipulation to the next via registers, thereby avoiding accessing memory for data. When the C code in the examples is compiled to run on a RISC architecture, the data manipulation steps result in the machine instructions noted in the comments. Hence, integrating the for-loops results in the elimination of one load and one store per word of data. Integration also reduces the loop overhead, since the integrated data manipulations share a single loop.

The principle underlying this example is that it can be more efficient to pass data between layers in units small enough to bypass memory, than to pass data in a large block via memory. More abstractly, memory hierarchies are optimized for locality of reference, and integration restructures a computation with poor locality of data reference into one with good locality. Thus, we propose the following definition: an implementation of a series of protocols is said to be *integrated* if the processing of a message by successive protocols is overlapped to achieve good locality of data reference.

The transfer of control between integrated layers must be highly efficient because it occurs with the passing of each data unit. Control transfer in this example is simple sequencing of instructions—trivial, but fast. Hence, the key elements of integration are passing small units of data, and transferring control efficiently.

Integration of data manipulations was previously investigated by Clark and Tennenhouse [4]. Their integrated data manipulations were hand-coded loops which performed all data manipulations in a single step while a data unit remained in register. The unintegrated versions were hand-coded loops with only one data manipulation per loop. Measurements on the MIPS R2000 processor showed unrolled loop implementations of checksum and copy processed data at 115 and 130 Mbits/second (Mbps), respectively. If these were combined in series, and none of the data were cached between data manipulations, the combined bandwidth would be 61 Mbps, while an unrolled loop that performed both operations at once ran at 90 Mbps. (Two other instances of folding checksum and copy operations into a single loop are described in [3].) In another test case, a routine to convert an array of integers into ASN.1 ran at 28 Mbps. When combined in series with the checksum manipulation, the combined bandwidth would be about 22.5 Mbps if none of the data were cached between data manipulations. Performing ASN.1 conversion and checksumming in a single step gave a bandwidth of about 24 Mbps.

Gunningberg, et al. [7] investigated integration further, using more interesting data manipulations and incorporating message header writing. The three data manipulations they considered were a simple presentation encoding, checksumming, and DES encryption. These data manipulations offer some complications not seen in the preceding examples: their presentation encoding increases the size of the data by inserting bytes, and DES encryption inherently processes eight bytes at a time. In the integrated form, the first layer (the presentation encoding) reads from a message buffer until it can output eight bytes; the data is subsequently passed between layers eight bytes at a time, in a pair of registers. A message header was generated by a technique customized to these particular data manipulations. The integrated version was compared with a strictly layered version in which the cache was flushed between data manipulations The integrated version gave only a 0.5% increase in bandwidth over the strictly layered version on a Sun SPARC station. The authors reason that the relative improvement was small because the DES algorithm is so slow (less than 1% of the bandwidth of each of the other two data manipulations) that it dominated the overall bandwidth. Computationally intensive data manipulations such as DES can be expected to benefit more from integration as processor speeds increase relative to memory speeds.

We make three observations about this previous work. First, the preceding examples are ultimately just examples—they suggest important techniques, but since the techniques are tailored to the particular data manipulations, their generality is limited. Moreover, they do not address all the elements of a complete message processing operation. In contrast, this paper presents a solution that is general and encompasses complete message processing operations. Second, both studies assume that no message data remains cached between data manipulations in the strictly layered case. In practice however, while caches are not highly effective for message processing, neither are they completely ineffective. To account for a range of possible cache effectiveness, this paper considers integration at both of the extremes: when all the data remains cached, and when none remains cached. Third, processor-intensive data manipulations like DES minimize the importance of avoiding memory accesses, and as a result, minimize the performance advantage of integration. We note, however, that most data manipulations have a low processing-to-memory ratio; and furthermore, that as processors become faster, even data manipulations like DES will begin to feel the effects of memory accesses. Thus, this paper considers the effects of integration in the limit.

1.2 Motivation for Automation

A program structure for integrated protocols is useful as a recipe for customized implementations, but our ultimate goal in devising such a structure is automating the integration process. Automated integration offers two important advantages over customized implementations.

First, automating integration adds the advantages of modularity to the inherent performance benefit. Each protocol is developed, debugged, maintained, and modified independently of other protocols. Furthermore, automated integration makes integration more scalable: the complexity of implementing a series of modular protocols is linear in the number of protocols (each protocol's implementation depends only on itself), rather than exponential, as in the case of monolithic implementations (each protocol's implementation potentially depends on each other protocol's implementation).

The second advantage of automating integration is reuse of protocol implementations. Clark and Tennenhouse [4] make the following observation: "A potential drawback to the ILP approach is that it could lead to complex designs in which each protocol stack variant has its own fully customized implementation. Clearly this would complicate the maintainability and overall utility of the protocol software." This prospect is even more ominous if one constructs network software from simple building-block protocols whose implementations can be reused in arbitrary compositions [10, 14, 2]. Automated integration eliminates this problem by supporting reuse of protocol implementations. A single modular implementation of a protocol can be automatically integrated into any number of protocol stacks.

Our concerns about reusable, flexibly-composed implementations performing competitively with integrated implementations were the original impetus for our research in automated integration. We concluded that keeping the benefits of modularity and reusability while gaining the performance benefit of integration would require automating integration. At the same time, our predisposition toward flexible composition of reusable modules was itself suggestive of an approach to automatic integration. Flexible composition of protocols hinges on a uniform interface between protocols. If the parts of an integrated implementation that belong to different protocols likewise conformed to a uniform interface, then separating them at the interface would result in protocol modules that could be composed in other ways.

1.3 Design Overview

Protocol integration is an optimization in search of a language. In an ideal world, this optimization would be performed by a compiler of one's favorite language. Protocol source code would be layered, but the compiler would generate object code in which the protocols were integrated. Unfortunately, compiler technology has not yet progressed this far. In our design, protocols are written in as fragments of C, and the "optimization" is applied in a separate pass to generate an integrated implementation in C. The integrated implementations are synthesized by interleaving code from the component protocols together with system-provided code for reading and writing message data structures.

The message processing operations in our protocol architecture are **send** and **deliver**. **Send** behaves as expected: a higher-level protocol passes a message to a lower-level protocol by invoking the a lower-level protocol's **send**. **Deliver** is similar, but is invoked (on the receiving side) by a lower-level protocol to pass a message to a higher-level protocol. Thus, **send** and **deliver** play identical roles except for direction: down versus up. While the presentation of our design is usually phrased in terms of **send**, it also applies to **deliver**.

An integrated **send** consists of three components: an *initial stage*, a *data manipulation stage*, and a *final stage*. All data manipulations are performed during the data manipulation stage, while the initial and final stages perform any other message processing tasks such as reading and writing headers, updating protocol state, and sending acknowledgement messages. In an integrated **send**, the initial stages are executed in sequence, then the data manipulations take place concurrently in one integrated stage, and then the final stages are executed in sequence. The semantics of the stages is that anything prior to the final stage is just "peeking" at the message; logically, a layer hasn't been passed the message until its final stage commences.

Each protocol's data manipulation stage is structured as a filter. This filter is a module that accepts as input a machine word of data, performs that protocol's data manipulation on the word, and passes the resulting data to the next protocol's filter one word at a time. The filter's code is inserted inline, and data words are passed using a shared variable implemented as a register. Integrated data manipulation consists of streaming message data through a pipeline of these filters.

This overview has described our design from the top down. The remainder of the paper presents the design (in more detail) from the bottom up since higher levels of the design follow from the decisions made at lower levels. Specifically, Section 2 describes a scheme for integrating data manipulations independently of the rest of a protocol, and Section 3 analyzes its performance. Section 4 then presents a message abstraction that supports our data manipulation scheme, Section 5 defines a structure for integrating complete **send** and **deliver** operations, and Section 6 presents a technique for automating integration. Finally, Section 7 discusses barriers to integration and Section 8 makes some concluding remarks.

2 Integrating Data Manipulations

At the heart of a design for integrating protocols is a design for integrating data manipulations. We temporarily make two simplifying assumptions to sharpen the focus on data manipulation. First, we pretend that data manipulation is all there is to processing a message, i.e., we ignore header processing and connection state. Second, we pretend that the message data originates from an array, and is finally written (if at all) to an array.

Each integrated data manipulation is structured as a data filter that processes one word of data at a time. In the common case, a filter outputs one word each time a word is input. However, some data manipulations change the amount of data (e.g., data compression), so one word of input can result in zero, one, or more words of output. An infrastructure-provided filter reads the array containing the original message data and invokes the first data manipulation's filter, once per word from the array. Each filter invokes the next data manipulation's filter when it has a word to output. A final infrastructure filter writes any output into an array. Filter invocation is realized by inserting the code for the filter inline, and data words are passed between filters via a C variable that is shared by all the filters and implemented as a register.

2.1 Data Units

Different units of data are optimal for different data manipulations. As a compromise, our approach uses the machine word (commonly 32 bits) as the unit for passing data between layers. Passing data in a fixed-size unit simplifies the interface between protocols. Our synthesis technique relies on all protocols having identical interfaces. Fixed-size units are also more efficient since variable-sized units would entail runtime interpretation or management.

One reason that the word is the optimal size for a fixed-size unit is that machine architectures are optimized for words. Arithmetic and logic instructions on RISC processors operate on registers, which are word-sized. Also, memory loads and stores of words are most efficient; partial loads and stores are slower. Passing a word at a time makes it possible to use a register, avoiding even the cost of cache hits. Because of the architectural bias for words, high-level languages tend to implement data types as words, especially the workhorse integer data type. In particular, local variables of word-sized data types can be implemented as register variables. This feature allows protocols to be integrated in a high-level language and still take advantage of passing data words via a register.

The other reason that the word is the optimal size for a fixed-size unit is that most data manipulations can efficiently process words. TCP/IP checksumming and byte swapping are half-word oriented but have efficient word oriented filters as illustrated by CKSUM (Figure 2(a)) and BSWAP (Figure 2(b)). Data manipulations based on units greater than a word are unusual but do exist. The Data Encryption Standard (DES) [13] is an example since it processes data in 64-bit (2-word) units. However, a filter implementing DES can accommodate 1-word units of data by saving the first word of each pair until the second word is received, i.e., by maintaining local state. DES exhibits a number of interesting data manipulation characteristics, but unfortunately software implementations of DES are so slow that, unlike other data manipulations, the computation time dominates the data access time. For this reason we have contrived an artificial data manipulation called *PES* (for *Pseudo Encryption Standard*) to investigate the data access characteristics of DES without the extensive computation.[2] Figure 2(c) uses PES to illustrate the use of local state to accommodate data in units smaller than those needed by the data manipulation. Note that local state is already required for many data manipulations, such as the partial sum in a CKSUM.

In our design, data is always passed one word at a time; fractions of a word are not permitted. This constitutes a constraint on data manipulations, but satisfying this constraint is simple. To maintain the integral-words invariant, any data manipulation that would modify an integral number of words of data into

[2]Think of PES as equivalent to DES running on a very fast processor.

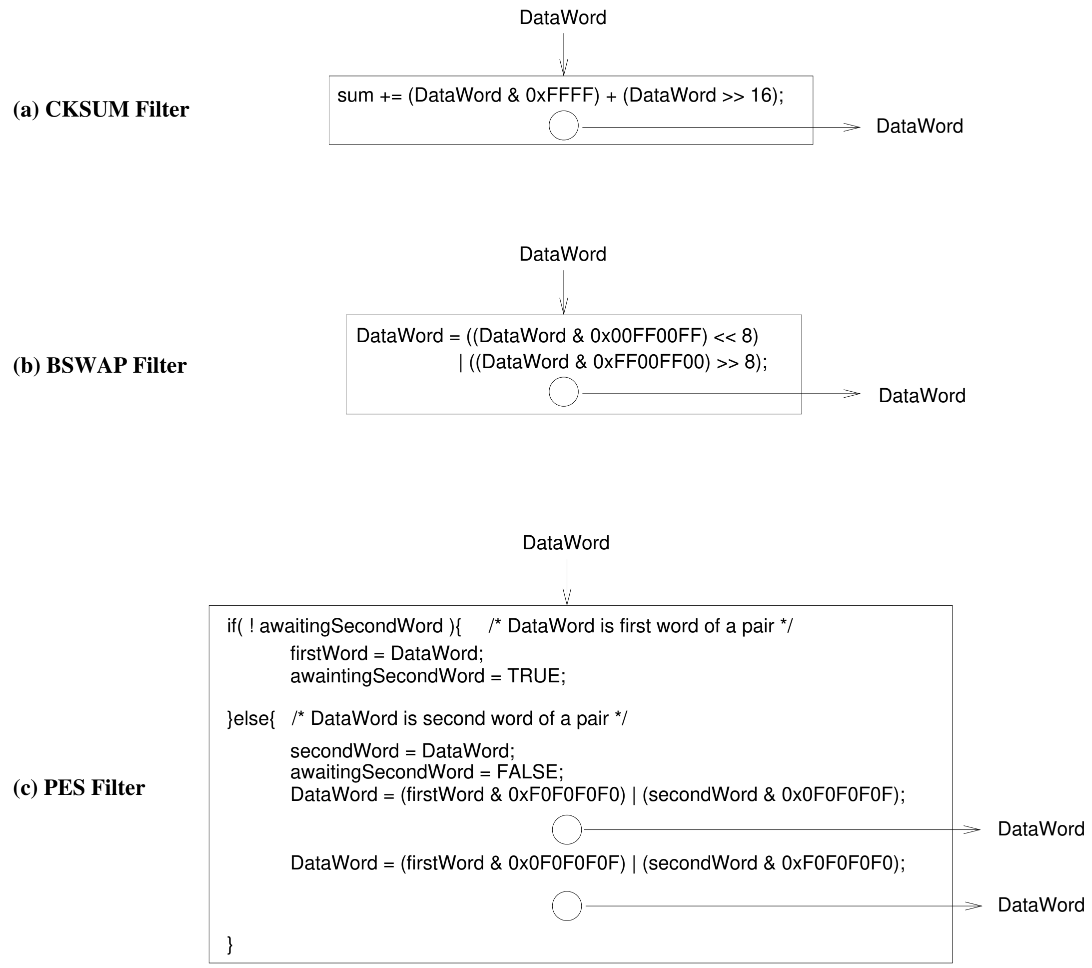


Figure 2: Filters

a non-integral number must pad its output the necessary fraction of a word. Moreover, being able to rely on an integral number of words makes many data manipulations simpler, since frequently they are optimized to operate on aligned words but have to special-case the odd bytes at either end of a data buffer (e.g., bcopy).

Filters behave like functions, but implementing filters as conventional functions would have two problems. The first is function call overhead: a function call is too high a price to pay each time a word of data is passed from one data manipulation to the next. Hence, we insert filters inline. The second problem is that each filter may need to maintain state across all its invocations corresponding to a particular message. For example, CKSUM's partial sum must be accumulated across invocations of the CKSUM filter until an entire message has been processed. If filters were implemented as functions, such state would have to persist as global or static variables to avoid passing it as arguments to each invocation. Normally globals and statics are implemented in heap storage. In contrast, implementing filters as inline code makes it possible to use local variables (local to the function into which a filter is inserted) to hold filter state, permitting the compiler to safely implement those local variables in registers.

2.2 Data Rates

If each data manipulation produced one word of output data each time it manipulated one word of input data, they could be concatenated in a simple for-loop. Unfortunately, data manipulations can change the amount of data. For example, a data compression filter generally compresses messages being sent and decompresses messages being received. More generally, data manipulations can produce data at a rate different from that at which they consume it, possibly without changing the total amount of data (e.g., PES).

In other words, a data manipulation may require decoupling its rate of input data consumption from its rate of output data production. This is accomplished by using state in the filter, and by language constructs such as conditionals and loops. For example, PES uses an if-else statement, while data decompression uses a for-loop. One consequence of this approach is that even though there may be no more input for a particular data manipulation filter, it may still have some output implicit in its state. Such output must be passed on to yet subsequent data manipulations. To ensure that this happens, when a given data manipulation has no more input, that manipulation is given the opportunity to output, or *flush*, any data that may be implicit in its state, passing it to subsequent layers. Flushing is illustrated using PES in Figure 3. Once a data manipulation has flushed, then the next manipulation can flush, and so on, until the last manipulation is done and there is no more output forthcoming from the entire integrated series of manipulations. Flushing only occurs in data manipulations whose output rate differs from their input rate, and not even all of those.

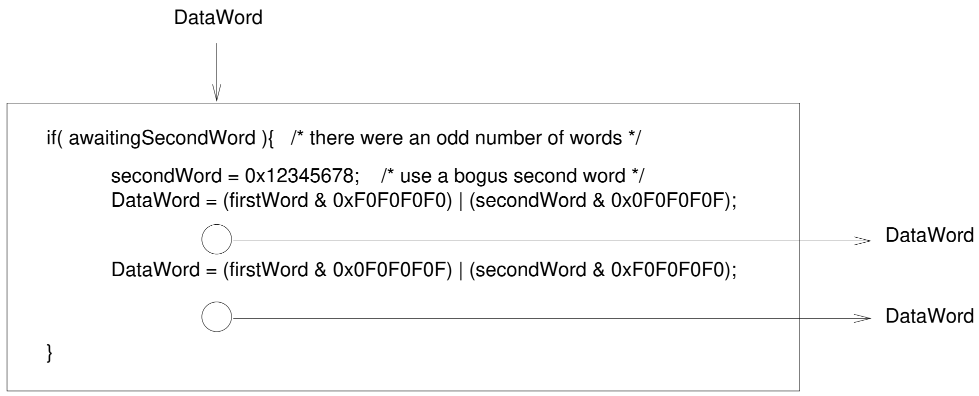


Figure 3: PES Flush

The requirement to support differing consumption and production rates is suggestive of Unix pipes, but this is misleading. The primary purpose of pipes is to decouple the rate of production at one end of a pipe from the rate of consumption at the other end. This is the opposite of what we want in integrated data manipulations. In terms of pipes, we want to use a very small pipe buffer (just one word) so that the buffer can be realized as a register. Moreover, we must couple the output rate of one data manipulation to the input rate of the next to avoid any synchronization cost, since such a cost would be paid each time a word of data is passed between a pair of data manipulations.

At each place where a filter or a flush outputs a data word, the subsequent filter is inserted inline. This is illustrated in Figure 4, using the combination of BSWAP, PES, and CKSUM. It may appear that the text of each filter is followed by the next, but more generally each filter *contains* the subsequent filters within its text. This is more apparent when a filter uses control constructs other than simple sequencing, as does PES. Passing a data word consists of leaving the word in the designated local variable and executing—in line—the subsequent filter, which may itself contain yet other filters. Thus, the "control transfer" between data manipulations is simple sequencing of instructions.

```
/* READ A WORD OF INPUT */
DataWord = *inputBuffer++;

    /* BSWAP */
    DataWord = ((DataWord & 0x00FF00FF) << 8) | ((DataWord & 0xFF00FF00) >> 8);

        /* PES */
        if( ! awaitingSecondWord ){     /* DataWord is first word of a pair */
                firstWord = DataWord;
                awaintingSecondWord = TRUE;

        }else{   /* DataWord is second word of a pair */
                secondWord = DataWord;
                awaitingSecondWord = FALSE;
                DataWord = (firstWord & 0xF0F0F0F0) | (secondWord & 0x0F0F0F0F);

                /* CKSUM */
                sum += (DataWord & 0xFFFF) + (DataWord >> 16);

                    /* WRITE A WORD OF OUTPUT */
                    *outputBuffer++ = DataWord;

                DataWord = (firstWord & 0x0F0F0F0F) | (secondWord & 0xF0F0F0F0);

                /* CKSUM */
                sum += (DataWord & 0xFFFF) + (DataWord >> 16);

                    /* WRITE A WORD OF OUTPUT */
                    *outputBuffer++ = DataWord;
        }
```

Figure 4: Combined Filters

2.3 Alternative Filter-to-Filter Interface

In the design just presented, integrated filters share a single register variable for passing data words. In fact, different variables could be used to pass words at different points in the text of the filters. Figure 5 presents a simpler and more efficient PES filter based on this technique. In this new filter, it is no longer necessary to assign the second input word to a temporary variable so that the input variable can be used for output.

Instead, the input variable is used directly and another variable is used for output. This saves one machine instruction on every second input word, an average of 0.5 instructions per word.

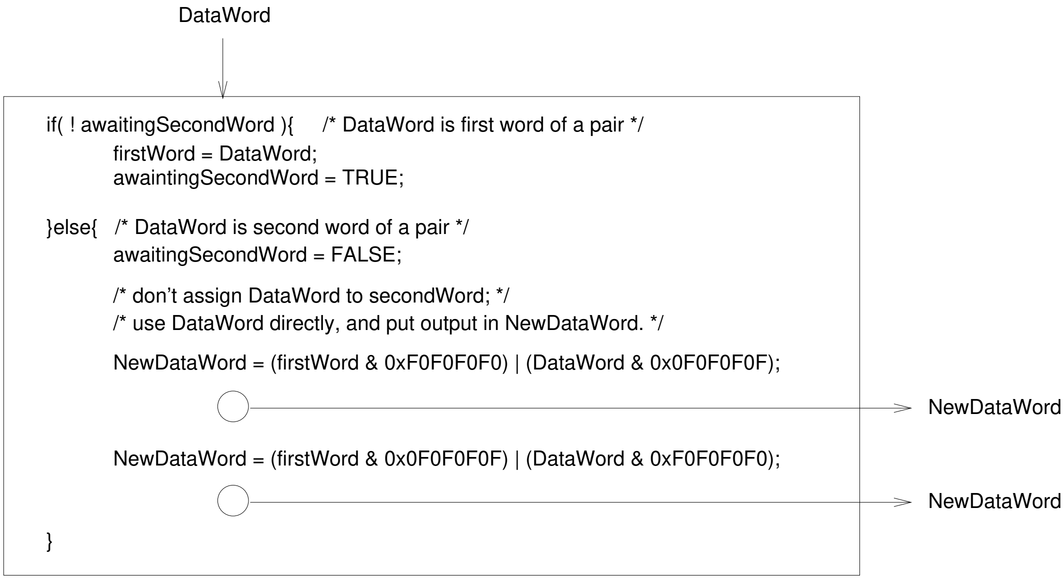


Figure 5: PES Filter Using Two Variables to Pass Data

This technique requires that the filters sharing a given variable agree on the name of the variable. This is easily arranged since, as is discussed in Section 6, we propose to insert filters inline at the source code level. Filters can be written using some canonical name for the input data variable, and when a filter is inserted inline, the textual name can be changed to match that of the output variable actually used by the outputting filter. For peak efficiency, all such data passing variables should be implemented as registers. If, as in the case of PES, using such a variable results in eliminating a local state variable (which should also be a register variable), then there is no extra demand on the register set. The remainder of this paper assumes the original technique, in which a single register variable is shared by all the integrated filters.

3 Measurements and Analysis

Still focusing on data manipulations under the assumptions outlined in the previous section, this section reports the results of experiments comparing serial to integrated implementations, and investigates the limits of integration. Each experiment was conducted on three different machines—the DecStation 5000/200, the HP 720, and the SPARCstation 1—to verify that our integration technique is not specific to a particular machine or cache design.

The MIPS R3000-based DecStation 5000/200 has separate 64-KByte instruction and data caches. The caches are direct mapped using physical addresses, and cache lines are 16 bytes. Loading a word from cache costs one cycle, and entails a one cycle delay slot. Storing a word normally takes a single cycle since the data cache is write-through, with a write buffer that can buffer stores for up to six pending writes and retire writes within the same page once per cycle. On this machine we used the Ultrix C compiler with the -O option.

The HP 720 has a 128-KByte instruction cache, and a 256-KByte copyback data cache. The caches are direct mapped and virtually addressed, and cache lines are 32 bytes. On this machine we used the HP-UX C compiler with the +O3 and +OS options.

The SPARCstation 1 has a 64-KByte combined write-through cache. It is virtually addressed, with a line size of 16 bytes. If the referenced line is in cache, loading a single word takes two cycles, and storing a single word takes three cycles. Several stores in succession will cause a stall once the memory write buffers fill, even if the target addresses are all cached. If the referenced word is not in cache, a single word load takes 14 cycles, and a single word store takes six cycles. On this machine we used SunOS C compiler with the -O option.

3.1 Case Study

To better understand the fundamental performance characteristics of integration, we timed different combinations of CKSUM, BSWAP, and PES. We chose these three data manipulations because they are representative of actual protocols. They also provide examples of both read-only and read-write data manipulations, and of both simple and convoluted filters.

The speedup obtained from integration depends on cache behavior, which in turn depends not only on the particular data manipulations, but also on the machine, the location of the message data in memory, and any concurrent processing. We isolated cache effects in our experiments by measuring performance at each of the two extremes: when all the message data is in the cache, and when none is in the cache. Thus, the actual performance in any particular situation must be within the range specified by the pair of measurements. To obtain cache hits, we simply ran the data manipulations back-to-back with no concurrent processing, using buffers sized and situated so as to avoid collisions in the cache. To obtain cache misses, we flushed the pertinent part of the cache between data manipulations, then subtracted out the time to flush.

Tables 1 and 2 give the results when the data is and is not cached, respectively. These tables show, for example, that on the DecStation 5000, integrating CKSUM and BSWAP increases bandwidth from 44.7 Mbps to 54.4 Mbps if it is cached, and from 29.3 Mbps to 39.6 Mbps if all message data is uncached.

	DS5000		HP720		Sparc1	
	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)
CKSUM+BSWAP	44.7	54.4	84.2	101.0	24.0	30.0
BSWAP+PES	33.9	43.5	72.1	101.0	17.1	25.0
BSWAP+PES+CKSUM	25.8	35.4	54.1	79.7	13.6	21.1

Table 1: Serial vs Integrated When Data in Cache.

	DS5000		HP720		Sparc1	
	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)	Serial (Mbps)	Integrated (Mbps)
CKSUM+BSWAP	29.3	39.6	42.4	55.8	19.3	26.1
BSWAP+PES	23.4	34.0	33.1	56.2	14.6	22.2
BSWAP+PES+CKSUM	17.5	29.0	26.2	48.9	11.3	19.0

Table 2: Serial vs Integrated When Data Not in Cache.

The throughput improvements resulting from integration are summarized in Figure 3. The table shows, for example, that the integrated implementation of CKSUM and BSWAP has 22% to 35% higher bandwidth than the serial implementation. The low end of each range (22% in this example) applies when all of the message data is cached, while the high end applies when none is cached. One would expect the improvment achieved in practice to fall between these two numbers.

	DS5000 (percent)	HP720 (percent)	Sparc1 (percent)
CKSUM+BSWAP	22 – 35	20 – 32	25 – 35
BSWAP+PES	28 – 45	40 – 70	46 – 52
BSWAP+PES+CKSUM	37 – 66	47 – 87	55 – 68

Table 3: Bandwidth Improvement due to Integration.

Several factors contribute to integration improving throughput. Most obviously, integration eliminates the load and store instructions themselves. Moreover, for those load and store instructions that remain, the delay slots after loads are more likely to be filled since there are more instructions available within an iteration, and similarly, on machines with write-through caches and write buffers, stores are less likely to fill a write buffer since they occur at a lower rate (i.e., more instructions are executed between consecutive stores). Another factor is that loop overhead is reduced since the integrated data manipulations share a single loop through the data, instead of iterating through the message data once for each data manipulation. Finally, integration eliminates some buffer allocations and deallocations since data which would have been buffered between data manipulations is now streamed directly from one to the next.

To verify that we have accounted for the most important factors in improved performance, we compared the time saved by integrating to the time taken by loops that simply load and/or store one word of data per iteration. The results are presented in Table 4, which for the sake of brevity, contains results for the DecStation only. These particular combinations of data manipulations give one example each of eliminating a load, a load and a store, and two loads and a store. The table shows, for example, that integrating CKSUM and BSWAP reduces the time to process one MByte of data by about 0.071 seconds if none of the data is cached, which is roughly equal to the 0.082 seconds it takes to load one MByte of data from main memory. Context-dependent factors, such as unfilled delay slots, could account for the small discrepancy.

The performance of an integrated implementation can suffer if a data manipulation is ill-suited to being

	Cache Hit		Cache Miss	
	Improvement (sec/MB)	Comparison (sec/MB)	Improvement (sec/MB)	Comparison (sec/MB)
CKSUM+BSWAP	.032	Load=.042	.071	Load=.082
BSWAP+PES	.052	Load+Store=.053	.107	Load+Store=.097
BSWAP+PES+CKSUM	.084	2×Load+Store=.095	.180	2×Load+Store=.179

Table 4: Comparison of Integration Savings.

structured as a filter. While this is not the common case, PES is to some degree an example, due to the complication of processing two words at a time. We implemented a strictly layered version of PES which is simpler and faster than our filter implementation because it is able to read and write two words at a time. We timed serial implementations using this faster PES and compared the results to integrated implementations using the filter PES. The integrated implementations were still faster in all cases except one where they were equal (BSWAP + PES, on the DecStation, in the cache hit case). Moreover, the stream-filter version of PES could be made slightly faster by using the alternative filter-to-filter interface in which a filter's output variable is different from its input variable.

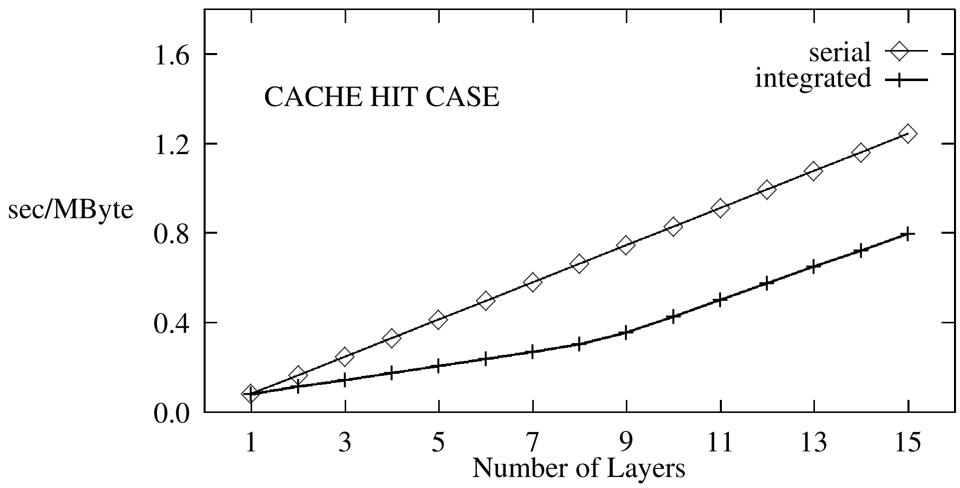
3.2 Scalability

We now consider how well integration scales, where the key limiting factor is register pressure. The central observation is that combining multiple data manipulations in a single function increases the demand for the registers for heavily accessed variables and constants. Consequently, integrated implementations may spill registers more than sequential implementations where each data manipulation has the entire register set to itself.

We conducted an experiment to investigate how integration scales by combining varying numbers of layers, where each layer performed the identical data manipulation but used its own two local variables. The data manipulation is a simple artificial data manipulation, intended to represent a generic data manipulation. The results for the DecStation are presented in Figure 6.

The slopes of the lines representing the serial implementations are constant because each additional layer adds the same amount of time. The slopes of the lines representing the integrated implementations change due to register pressure. Each layer of data manipulation uses its own two local variables. When only a few layers are integrated, these variables are all implemented as registers. As additional layers are integrated, the additional variables must be implemented on the stack. Hence, the cost of an additional layer is smaller when there are still registers available for variables, and greater when there are more variables than usable registers. The knee in the curves representing integrated implementations occur at the point where the number of variables exceeds the available registers.

At the knee in the DecStation case (eight layers), the integrated version runs 2.2 times faster than the serial in the cached case, and 3.0 times faster in the uncached case. Table 5 reports where the knee occurs



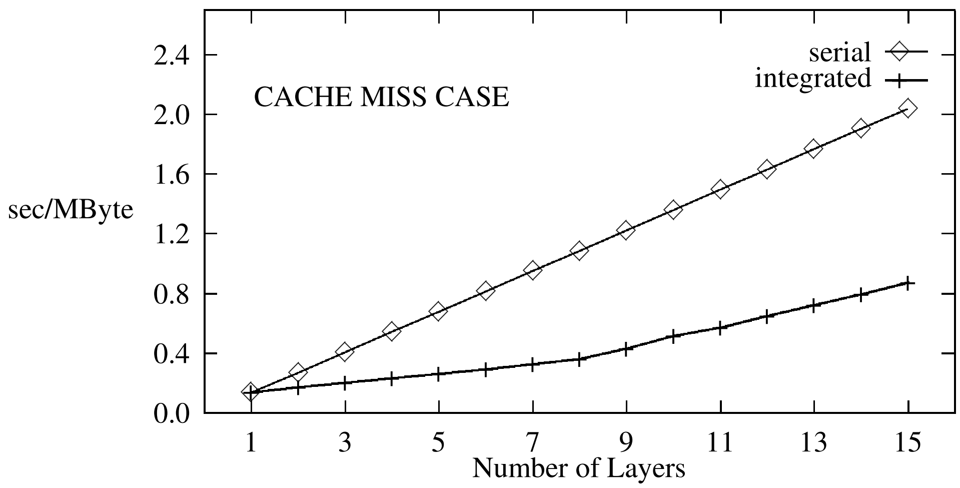


Figure 6: Incremental Performance On DecStation

and the associated improvments factors for the three machines we tested. Notice that the difference in the improvement factor in the cached and uncached case becomes more dramatic as the processor becomes faster—in this case, the HP720 is faster than the DS5000, which is faster than the Sparc1.

	DS5000	HP720	Sparc1
Knee (number of layers)	8	7	5
Improvement ($\times$ faster)	2.2 – 3.0	2.0 – 3.8	2.6 – 2.8

Table 5: Improvement Factor at Knee of the Integration Curve.

Consider the tradeoff that holds between integrated and serial implementations once the knee of the curve has been exceeded. In the serial implementation, local variables stay in registers while message data access involves loads and stores. The opposite holds in the integrated implementation; message data stays in a register, and it is local variable access which involves loads and stores. Simply comparing the number of loads and stores is not sufficient to determine which implementation is faster, since the speed of loads and stores depends on caching. Variables implemented on the stack remain in cache a high percentage of the time since they exhibit good locality. If message data tends to be uncached, then the integrated implementation can afford many (cached) accesses to local variables to avoid (uncached) accesses to memory data. If instead message data tends to remain cached, loading and storing a word of message data is as fast as loading and storing a one-word variable on the stack. This makes it possible for a serial implementation to outperform an integrated implementation—given enough local variables per layer and a sufficiently high message data hit rate.

By extrapolating our measurements, we determined that two read-write local variables per layer and a 100% hit rate for message data would be sufficient for the serial implementation to outperform the integrated implementation on the HP 720 and Sparc1 if the number of layers grows beyond 33 and 23, respectively. On the DecStation, however, the slope of the integration curve in both the cached and uncached cases is less than the slope of the serial curve, meaning that the serial implementation never outperforms the integrated implementation. One possible reason that the integrated implementation remains faster on the DecStation is that the high speed of its cache-hit loads and write-buffered stores relative to other operations makes the loop overhead eliminated per layer by integration more significant.

Obviously, 20 – 30 layers is more than one would integrate in practice. On the other hand, our experiments clearly demonstrate that additional registers would benefit integration as the number of layers increase.

3.3 Space

Inserting filters inline has the potential to use space exponential in the number of integrated data manipulations because each filter's code may be inserted in more than one place in the preceding filter. This is unlikely to be a problem in practice because the number of integrated data manipulations will be modest, and most filters are invoked in only one place in the preceding filter. If it were to become a problem, space consumption could be reduced by breaking a long series of integrated data manipulations into multiple shorter series. A potential space problem due to applications is discussed in Section 4.2.

Conceivably, combining code from several data manipulations could have a negative effect on caching due to the decreased locality of reference to instructions. Integration leads to iterating through the code belonging to a whole series of data manipulations, instead of iterating through the smaller bodies of code representing an individual data manipulations. Instruction locality may be further reduced by duplicating a data manipulation inline at multiple points in the program text. To some extent, integration trades locality of instruction references for locality of data references. Yet, this larger working set of instructions is still very small compared to current cache sizes such as the DecStation 5000/200's 64-KByte instruction cache, the HP 720's 128-KByte instruction cache, and the Sparc1's 64-KByte combined cache.

We have not observed any degradation attributable to inlining in our experiments integrating up to 15 (simple) layers. In general, inlining seems to have little effect on caching and virtual memory. Experiments reported in [5] showed no obvious evidence of either instruction cache overflow or thrashing, and the previous reports they cited showed similar results.

4 Message Abstraction

We have thus far discussed data manipulations as if they could be isolated from other message processing activities such as reading and writing headers. This is an oversimplification since in the conventional treatment of messages, one protocol layer's header is just more opaque data to lower layers. We propose an alternative abstraction that explicitly separates protocol headers and application data, thereby separating data manipulation from header processing. We refer to the conventional message abstraction and our alternative abstraction as *encapsulated* and *segregated* messages, respectively, and illustrate them in Figure 7. The key idea is that segregated messages support higher performance integration than the encapsulated messages, while retaining convenient random access to header fields.

4.1 Encapsulated Versus Segregated Messages

When a protocol sends a message, it may prepend a header or append a trailer which its receiving peer knows how to interpret. Lower protocol layers using the conventional encapsulated messages do not distinguish that header or trailer from the rest of the data—it is all opaque data. This is a form of hierarchical encapsulation. The purpose of hierarchical encapsulation is modularity, i.e., avoiding dependencies between protocols.

Hierarchical encapsulation complicates integration. The basic problem is that even though adjacent layers are to manipulate data concurrently, they have different definitions of the portion of the message that is data to be manipulated. Consider two particular situations. First, a sending protocol usually cannot write its header or trailer until its data manipulation is complete (e.g., CKSUM). This is a problem because any lower layer integrated with that protocol must manipulate its data concurrently, even though higher level headers are not available until afterward. Second, a receiving protocol does not necessarily know what to do with its data, or even what portion of the message is its data (e.g., variable length headers), until it reads

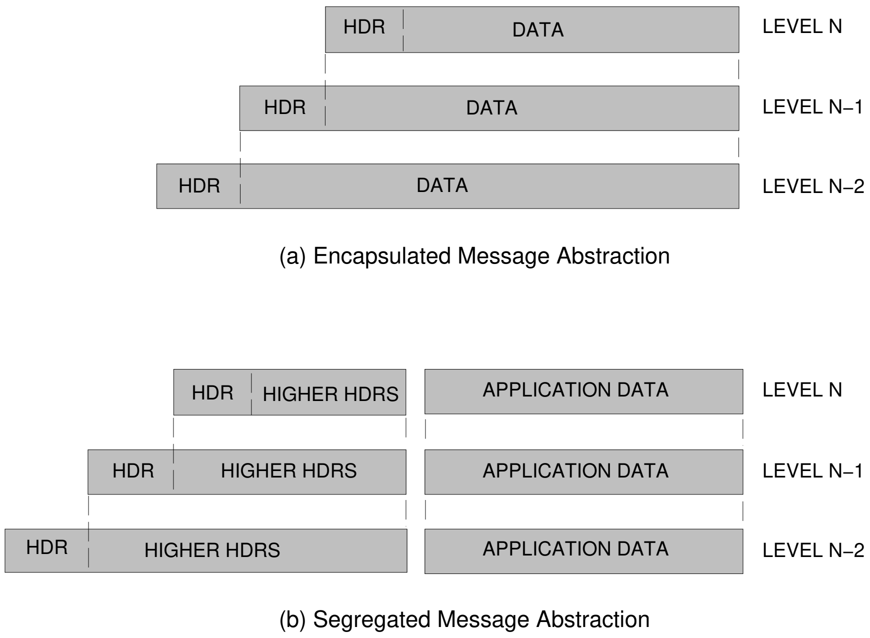


Figure 7: Message Abstractions

its header or trailer. This is a problem because that protocol must manipulate its data concurrently with any lower layer with which it is integrated, even though its header or trailer is part of the data that must be manipulated by the lower layer.

One solution would be for protocols to use only trailers on the sending side, so that each protocol's "header" information would be available before the subsequent layer needed to manipulate it. Receiving side protocols would then process a message "backwards," with trailers effectively becoming headers, so that receiving protocols would have their header information available before performing data manipulation. Headers would be added to the data stream the same way that leftover output is flushed from the state of a filter.

However, our alternative message abstraction offers better performance and programmability than this approach, yet preserves the modularity that is the motivation for strict hierarchical encapsulation. With segregated messages, the portion that is application data is distinguished from the portion comprising protocol headers. Because the contents of the two parts remain opaque to lower level protocols, this relaxation of strict hierarchical encapsulation nonetheless preserves modularity and avoids any dependencies between specific protocols.

Segregated messages support sequential processing of protocol headers independently of whether the manipulation of application data is integrated. Such messages have two coexistent implementations. For unintegrated protocols, messages are implemented as a conventional buffer-based data structure. For integrated protocols, messages are implemented as a combination of conventional data structure (for headers) and runtime data stream (for application data).

The distinction between headers and trailers disappears with segregated messages. There is only a stack of control information that is disjoint from a message's data payload, being neither at its head nor at its tail. For simplicity, we will continue to refer to this control information as "headers." Headers are implemented as a buffer-based data structure, even for integrated protocols. Since this permits random access to header fields, reading and writing headers can be programmed in a conventional style as opposed to the stream-filter style. Furthermore, this is more efficient than the stream-filter style since a receiving filter would have to execute an extra conditional per word of data to process its header differently from other message data.

Segregated messages permit the integrated manipulation of protocol data without the complication of headers. By the same token, any headers belonging to higher level protocols must be manipulated separately from the integrated manipulation of application data. This could result in additional programmer effort if the same data manipulation must be coded twice, once for application data and once for higher level headers.

We have concluded that the manipulation of higher level headers should not integrated. The potential performance improvement is insignificant due to the small size of headers. Furthermore, it would sacrifice random access to header fields, and use more space by duplicating filters.

A network device typically delivers a message as a sequence of bytes in the form of an uninterpreted list of buffers. The network device driver then elevates that list of buffers into a message data structure implementing conventional encapsulated messages. Elevating a list of buffers into a segregated message is only a little more complicated, since all that is required is rediscovering the demarcation between the combined headers and the application data. To accomplish this, a layer at the bottom of the protocol stack uses its header to record where the combined headers end and the application data starts. This layer could be implemented as part of the network device driver, or as a separate module on top of low-level protocols (e.g. IP) that use the conventional abstraction.

4.2 Sources and Sinks of Message Data

In our discussion of integrating data manipulations, we assumed that the original input data is obtained from an array, and likewise the final output data is written into an array. In the context of integrated protocols, the sources and sinks of message data are more complicated.

There are two possible sources for the message data processed by an integrated series of protocols. One source is a message data structure, the data structure realization of segregated messages. In this case, infrastructure code loads data words from the message data structure and passes them one word at a time to the filter of the first integrated layer. Loading the data is not quite as simple as iterating through an array since a message data structure generally involves a list of non-contiguous buffers [6, 9]. The reason for the list instead of a single contiguous buffer is to avoid copying data when fragments are reassembled, to allocate space for data incrementally when the total space required for a message is not known in advance, and to share parts of the physical data between logically different messages without copying.

The other possible source of data is an application that *generates* data. An application may generate message data directly from some computation, or by taking it from local variables, files, or input devices.

As before, data is passed one word at a time to the filter of the next integrated layer.

As with sources, there are two kinds of sinks for the message data produced by a series of integrated protocols. One kind of sink is the message data structure. In this case, infrastructure code in the form of a filter stores its input data words into a message data structure. The other sink for data is an application that *consumes* data. An application may consume message data by using it directly in some computation, or by writing it to local variables, files, or output devices, for example.

If the protocols in an integrated series get their data from a message data structure and are all read-only with respect to message data, then they produce no output data to sink; the input message data structure still has the current form of the data (although the protocols have changed the combined header). In that case, output from the last protocol's filter is ignored.

The interface between an application and an underlying filter is the same as the the interface between filters—an application is treated as a protocol. Hence data manipulation by an application can be integrated with lower level data manipulations. This is critical since data manipulations such as presentation formatting are often application-specific.

Integrating applications introduces the potential for excessive code duplication. An application could transmit irregular data using many textually distinct invocations of the next protocol's filter (e.g., an RPC stub marshalling scattered arguments). Duplicating the integrated protocol filters inline at each invocation point in the application could use excessive space. An application could avoid this by calling, from many places, a single function that implements the underlying collection of integrated layers.

5 Integrating Complete Send and Deliver Operations

Our design for integrating data manipulations was based on two unrealistic assumptions. The first was that data is read from an array and written to an array. The introduction of the new message abstraction and the discussion of sources and sinks of message data lifted this assumption. The second assumption was that there were no message processing tasks other than data manipulation. This section lifts that assumption and presents a design for integrating complete **send** and **deliver** operations. In this design, streamed message data originates from generators or message data structures and terminates at consumers or message data structures, and all message processing tasks, such as reading or writing headers and updating connection state, are accounted for.

The correctness of integrated protocols may be defined as consistency with the behavior of a serial implementation, although this may be too conservative[3]. Just as concurrent database transactions must be serializable, we require an integrated series of protocols to behave as would a strictly layered series of

[3]Integration introduces events, such as passing a chunk of a message between protocols, that either are not specified, or are implicitly ruled out by a specification. In the latter case, we are really introducing a standard of correctness for implementations of a *series* of protocols that supersedes correctness of individual protocols. In this view, the implementation of a *series* of protocols is correct if it satisfies the specification obtained by composing the specifications of the individual protocols and eliding the events that are now internal to the composite specification.

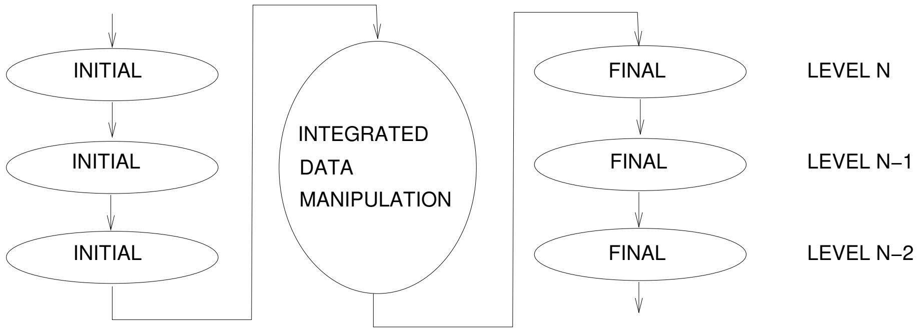


Figure 8: Execution Sequence of Integrated Protocol Stages

protocols. In the case of protocols, behavior includes not only externally visible actions such as **send**ing a message, but also state changes since they affect future actions.

Each protocol's **send** and **deliver** are split into three stages: the *initial stage*, the *data manipulation stage*, and the *final stage*. Tasks other than data manipulation take place during the initial and final stages. The data manipulation stage is integrated as previously described, but each layer's initial and final stages are executed atomically with respect to other layers. Given an integrated series of protocols, the initial stages are executed serially, then the integrated data manipulation structure is executed, and then the final stages are executed serially. This is illustrated in Figure 8.

Consider the particular tasks performed by protocols—integrated or not—aside from data manipulation. These include reading and writing headers; initializing data manipulation variables; updating protocol state; setting and clearing timers; sending control messages; passing non-message information (such as flow control) to adjacent layers; demultiplexing or making routing decisions, and (assuming segregated messages) performing data manipulation on the combined higher level headers. Of course, a given protocol need not perform all of these, or even any of them. For simplicity, we lump these into the following categories:

- Reading/writing own header.
- Manipulating higher level headers.
- Externally visible actions and updating protocol state.

In a strictly layered implementation of a protocol, the implementor knows that the protocol has access to a message only after all preceding layers are finished with it and before subsequent layers have access to it. Hence, the ordering of a given protocol's tasks is subject only to constraints internal to the protocol. An example of an internal ordering constraint involving a checksum protocol is that it cannot write the checksum into the message header until after it has read all the data. In contrast, protocol integration involves the concurrent processing of a message by multiple protocols. This introduces external constraints on the ordering of a protocol's tasks. An example of an external ordering constraint is that one protocol cannot manipulate the previous protocol's header until the previous protocol has written its header. (In a sense, strictly layered implementations are subject to one overriding external constraint: a protocol cannot access a message until the preceding protocol is completely finished.) These external constraints interact with internal constraints to fix particular tasks in particular stages, as listed in Table 6 and explained in the following subsections.

TASK	STAGE
write header (sending)	final
manipulate higher headers (sending)	final
read header (delivering)	initial
manipulate higher headers (delivering)	initial
update state	final
take externally visible action	final

Table 6: Each Task Must Be Executed In The Corresponding Stage

It is critical to be able to integrate across protocols that do not manipulate data. Otherwise, every such protocol would act as a barrier to integration, greatly reducing the potential performance improvement from using protocol integration. This would be especially debilitating for protocol suites composed of large numbers of simple building-block protocols, such as we have advocated elsewhere [14, 2]. In our design, protocols that do not manipulate data are structured similarly to data manipulation protocols, except their data manipulation filter is null.

5.1 Header Processing

Consider the constraints on the ordering of the header processing tasks—reading/writing headers and manipulating higher level headers. First, consider the sending side. A protocol's header may depend on the results of data manipulation (e.g., checksum), in which case it cannot be written until the protocol's final stage. Since no protocol can assume that the preceding protocol's header will be written any sooner than the preceding protocol's final stage, it cannot perform data manipulation of higher level headers until its own final stage. Furthermore, we stack consecutive headers in contiguous memory to make data manipulation of the combined headers simpler and more efficient. Hence, a protocol cannot know *where* to write its own header until the previous header has been written and possibly manipulated, meaning it cannot write its own header until its own final stage. In summary, because some protocols cannot write their headers until their final stage, all protocols defer writing headers and manipulating higher level headers to their final stages.

An analogous argument applies on the receiving side. In general, a protocol cannot perform data manipulation on a message without having first read its header from the message, and a protocol cannot read its header until the previous protocols have performed any data manipulations on the combined headers. A protocol cannot even determine where its header starts until the previous protocols have all consumed their headers if we again assume that consecutive headers are stacked in contiguous memory, Thus, because some protocols must read their headers in their initial stage, all protocols must read headers and manipulate higher level headers during their initial stages.

5.2 Message Rejection

A message may be rejected by a protocol for a variety of reasons; for example, it may be a duplicate, have an illegal format, or require unavailable resources. The reason may lie with the message (e.g., corrupted data) or with the protocol state (e.g., flow control). Message rejection is more common on the receiving side, but may also occur on the sending side, for example as a result of congestion control.

Message rejection is a potential problem for protocol integration because a message may be rejected "in the middle" of an integrated series of protocols. Layers which are logically subsequent to the rejecting layer may have already begun processing the message. Furthermore, layers which logically precede the rejecting layer may not have completed processing the message; in particular, it may be that in a sequential implementation, a preceding layer would have rejected the message before it ever arrived at the current rejecting layer! In other words, the problem is maintaining the semantics of a serial implementation with regard to message rejections. This problem is analogous to that of aborted or failed transactions among concurrent database transactions.

Intuitively, the solution is to treat the initial and data manipulation stages as if they were merely peeking at a message that is not really passed until the final stage. There are two aspects of realizing this solution. The first is to defer any message rejections to a protocol's final stage. That is, any protocol that has cause to reject a message during its initial or data manipulation stage must defer the rejection and continue to process the message. This gives logically preceding layers the opportunity to reject the message first. The final stages execute serially, so the rejecting layer that is logically earliest will be the layer that actually ends up rejecting the message.

The second element of this solution is to defer to the final stage any externally visible actions and any updating of protocol state. That is, having determined the correct rejecting layer, it is also necessary that the current and future behavior of the integrated protocols be consistent with the message having been rejected at that layer. For example, no subsequent layer should have acknowledged the message, or counted it against a flow control window. The final stage then serves as a *commit* stage, corresponding to the point in a sequential implementation when one layer passes a message to the next. The initial and data manipulation stages are then considered to be previewing or peeking at a message that hasn't been logically passed until the previous layer completes its commit stage.

This approach has a potential problem with data structures that must be explicitly deallocated, such as an output message data structure. Suppose a protocol allocates such a data structure in its initial stage, but its final stage is not executed because the message is rejected by a preceding protocol; then the data structure would never be deallocated. One approach is to add a fourth stage—the *abort stage*—that would provide the opportunity to deallocate data structures. When a message is rejected by an integrated protocol, the subsequent protocols in the integrated series would have their abort stages executed instead of their final stages. In this scheme, a rejection requires an active invocation of the next layer's abort stage; in the preceding scheme, message rejection is passive—the rejecting layer simply doesn't invoke the next layer's final stage.

5.3 Separate Control Path

Messages do not always contain application data; protocols often exchange *control messages* that consist entirely of protocol headers. Examples include acknowledgements, flow control messages, and connection establishment messages. Control packets are easily identifiable as any message that has no application data. Even on the receiving side, it is obvious whether a message holds application data because segregated messages separate application data from all protocol headers.

In our design, control messages are processed by operations distinct from those for application data messages. In other words, each protocol implements two **send**s and two **deliver**s: **sendControl** and **deliverControl** operations for control packets, and **sendData** and **deliverData** operations for data messages. The data path is integrated; the control path is not. Both paths manipulate shared connection state, however.

One motivation for this decision is to optimize message processing appropriately for each message. While bandwidth is the performance parameter of primary importance for data messages, with control messages we are more often concerned about latency. Splitting the message processing path permits us to optimize the data path for bandwidth while optimizing the control path for latency.

Another motivation for separate control and data paths is to support the origination and termination of messages at arbitrary layers. Data messages always originate and terminate at an application at the top of a protocol stack, but control packets such as connection establishment messages can originate and terminate at any layer. This poses a problem because hard-wiring together integrated layers in a single function does not permit intermediate layers to initiate or consume a message. A separate control message path eliminates this problem.

5.4 Performance of Integrated Protocols

We measured the performance of some protocols to see how well the benefits of integration carried over from data manipulation to complete message processing operations. The protocols were based on the CKSUM, BSWAP, and PES data manipulations measured in Section 3. The filter version of PES was used for both serial and integrated timing. We measured **send** operations, so the results represent bandwidth from the top of a series of protocols to the bottom. The measurements were made on the DecStation 5000/200. Table 7 shows that the integrated implementations were consistently faster than the serial implementations. For example, integrating the CKSUM and BSWAP protocols increased bandwidth from 34.6 Mbps to 40.6 Mbps, an improvement of about 17%. By way of comparison, integrating the CKSUM and BSWAP *data manipulations* gave a 22% to 35% improvement, as reported in Section 3.

These experiments used messages consisting of a single contiguous 4-KBbyte buffer. Additional experiments confirm that varying the number of buffers making up a message while holding the size of the message constant has little effect on performance as long as the buffers are reasonably large. These results represent a very high percentage of cache hits for the message data since the message size was modest (one page), there was no explicit flushing of the cache, and there were no context switches, nor intermediate protocol layers between the data manipulation protocols. In a real system, one would expect the cache to be

DATA MANIPULATIONS	SERIAL (Mbps)	INTEGRATED (Mbps)	IMPROVEMENT (% of serial)
CKSUM+BSWAP	34.6	40.6	17%
BSWAP+PES	22.9	29.5	29%
BSWAP+PES+CKSUM	18.8	25.6	36%

Table 7: Serial vs Integrated **send**s.

somewhat less effective, so that the performance gap between serial and integrated implementations would be somewhat greater.

The bandwidths reported here tend to be lower than even the cache miss cases measured for the corresponding isolated data manipulations. Yet, as discussed above, one would expect a high percentage of cache hits for the message data. One reason for this discrepancy is the overhead of all the protocol tasks other than the data manipulation, although this is essentially constant per message. Another reason is that our protocol development environment depends on the GNU C compiler (gcc) with the -O option instead of the Ultrix C compiler with the -O option which was used for the data manipulation experiments on the DecStation. Gcc generates less efficient code for these particular data manipulations. In particular, there are 32-bit masks used by BSWAP and PES, which gcc regenerates (at a cost of 2 cycles each) each time they are used instead of keeping them in registers like the Ultrix compiler. This adds four cycles per data word to each of BSWAP and PES.

PES was further slowed by 0.5 cycles per data word as a side effect of incorporating it in a protocol. PES must pad data to an even number of words. Discarding padding from a data stream—unlike a data structure—requires knowing where the padding starts. PES's **sendData** keeps a count of the number of words, so it can write this count in its header.

6 Synthesizing Integrated Protocols

Our design for integrated protocols achieves a number of goals. It meets the standards of correctness of a serial implementation, yet performs significantly better. It is reasonably independent of machine architecture and compiler, and supports a wide range of protocols. The main goal not yet explicitly addressed is synthesis: automatically generating integrated protocols from unintegrated, modular input protocols. In fact, the synthesis technique itself is simple; the work went into devising an integrated code structure that would accommodate it. This section illustrates the synthesis technique with a prototype that integrates input protocols expressed as textual fragments of C code.

Our design for automatic integration calls for integrated protocols to be output in a high-level language, C in the case of our prototype. The reason for a high-level target language is to isolate integration from other language implementation issues, allowing us to focus on integration in the design and its presentation. The compiler for the high-level language has the responsibility for taking advantage of the greater locality of reference exhibited by the integrated protocols, e.g., using registers to hold message data between data

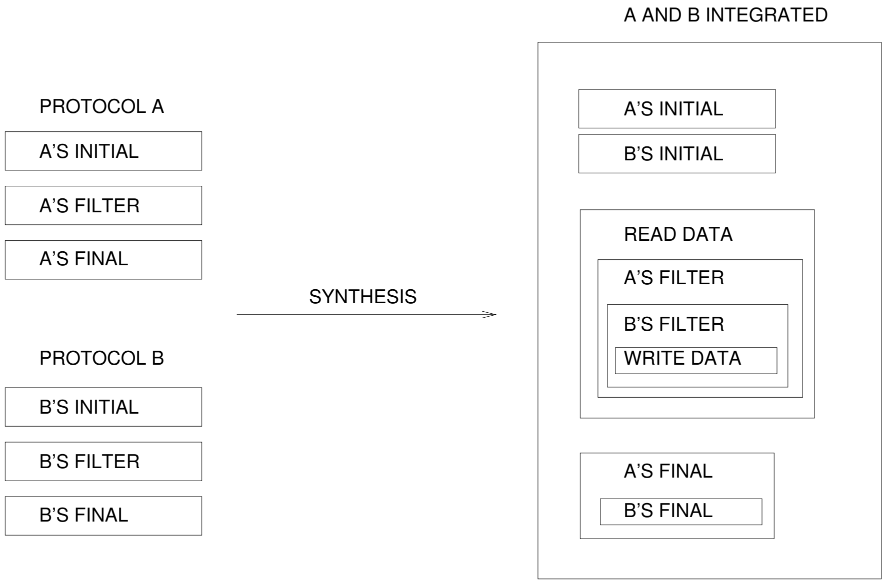


Figure 9: Integrated Protocol Synthesis

manipulations. In our experiments, the compilers have done so successfully, utilizing registers as expected.

This separation of language issues has a number of technical advantages. First, directly generating assembler code would require intertwining integration with other compiler tasks such as register allocation. Second, a separate integrator that outputs a standardized language is portable, since the compiler hides machine dependencies. Finally, operations other than the data message processing operations can be written directly in the target language.

Our structure for integrated protocols has the property that, aside from the code for reading and writing message data structures, it can be partitioned into disjoint code fragments, each belonging to just one component protocol. Each protocol's fragments conform to the same interface. Our synthesis technique simply reverses this partitioning. It concatenates input protocol code fragments in the proper order along with the code for reading and writing message data structures, as depicted in Figure 9.

Specifically, the prototype integrator is simply the macro processor *m4* [11] applied to appropriate input and include files. A data message processing operation (**sendData** or **deliverData**) corresponding to a given protocol is input as six code fragments. The CKSUM protocol is used as an example in Table 8. The INCLUSIONS fragment is for including header files and making type or macro definitions; the DECLARATIONS fragment is for declaring local variables; and the INITIAL and FINAL fragments correspond to the initial and final stages, respectively. The two remaining fragments correspond to the data manipulation itself: FILTER is the data filter invoked by the preceding protocol when it passes a word of data, and FLUSH provides a way to flush data saved in the local state to subsequent layers once there is no more data forthcoming from preceding layers. In the case of CKSUM, there is no code to execute for a flush. The FILTER and FLUSH fragments must indicate explicitly the points where they output data, to indicate where to inline the next protocol's filter. Similarly, the FINAL fragment must explicitly indicate the point at which it passes the message to the subsequent layer, since this need not be its very last action.

FRAGMENT PURPOSE	CHECKSUM EXAMPLE
INCLUSIONS	#include "chksum.h"
DECLARATIONS	cs_Session cs_session; cs_Hdr cs_hdr; register unsigned cs_sum;

INITIAL	cs_session = (cs_Session) next_session;
	next_session = (GenericSession) cs_session->nextSession;
	cs_sum = 0;

FILTER	cs_sum += (currWd & 0x0000FFFF) + (currWd >> 16); include(CS_NEXT/filter)
FLUSH	/* checksum has no flush */

FINAL	while(cs_sum & 0xFFFF0000)
	    cs_sum = (cs_sum & 0x0000FFFF) + (cs_sum >> 16);
	cs_hdr = (cs_Hdr) msgPush(hdrMsgP, cs_HDR_SZ);
	cs_hdr->chksum = cs_sum;
	include(CS_NEXT/finalStage)

Table 8: CKSUM **sendData** Code Fragments

The message data processed by integrated layers originates from either a message buffer, or an application that generates the data itself. If the data originates from a message buffer, the integrator generates code that reads the data from the message buffer and streams it to subsequent layers. This data-reading infrastructure serves as a framework in which all the protocol fragments are embedded inline, thereby establishing the sequencing of stages. When the data is generated by an application, the application streams the data directly to subsequent layers. In this case, the application itself serves as the framework in which the fragments are embedded.

Unless all the data manipulations integrated together are read-only, the data must finally be "consumed," either by being delivered to an application or by being written into a message buffer by code inserted by the integrator. The **deliverData** of a data-consuming application is structured like that of protocols except that it does not invoke any subsequent layers.

One major shortcoming of this prototype implementation is that because it uses programmer code verbatim, it requires the programmer to use names that do not clash with those used in other protocols. This includes not only variable names, but also names of types, defined constants, macros, functions, and labels. The problem arises because protocol integration has the effect of fusing some of the branches in a tree structured name space. The brute-force strategy we have used for prototyping is to assign a globally unique name to each protocol (e.g., "cs_" for CKSUM), and prefix that name to every name used by the protocol.

The other major shortcoming of this implementation is that mechanics of the synthesis technique are apparent in the idiosyncrasies of the input syntax. This input syntax forces the programmer to be aware of integration and the synthesis technique. Ideally, a new message processing abstraction should be expressed using intuitive language constructs that are consistent with the rest of the language. One possibility is to incorporate such an abstraction in the design of a new language such as the protocol implementation language Morpheus [2]. A more economical alternative is to extend an existing language (the output language) with new but compatible message processing constructs. Both approaches are discussed in more detail elsewhere [1].

7 Barriers to Integration

Automatically integrating protocols offers the best of both worlds: integrated protocols perform significantly better than serial implementations, while automatic synthesis restores the modularity benefits of serial implementations. This would suggest that all protocols should always be integrated. Unfortunately, there are situations in which it is better not to integrate. We have identified several classes of barriers to integration—cases where protocols should not or cannot be integrated:

Control Transfer If adjacent protocol layers are in different address spaces or different processors, transferring control between them is too slow for integration.

Message Reassembly Lost or out-of-order message fragments make it difficult for protocols above a reassembly layer to process isolated fragments.

Random Access Some protocols need random access to message data, and therefore, they find the sequential access provided by a data stream too inefficient.

Retransmission In some cases a new physical copy of a message must be buffered, but whenever possible it is better to share a message buffer associated with a different protocol layer.

Runtime Protocol Path For some protocols, there is more than one possible next protocol, such as when demultiplexing.

Some of these are due to how protocols are mapped onto address spaces, and some are inherent to protocols themselves.

There are strategies for minimizing barriers. A basic strategy is to avoid protocols that are barriers. When this is not possible, barriers may be further minimized by a careful ordering of protocols in the protocol graph, and a careful mapping of protocols onto protection domains. The key observation is that barriers are only barriers if they occur between data manipulation protocols, since the performance advantage comes from integrating data manipulations, not arbitrary protocols. We integrate only non-data manipulation protocols when they come between data manipulation protocols. Hence a barrier that is located either above or below any data manipulations does not prevent any integration. Also, if integration between two data manipulations has already been blocked by a barrier, then it does not matter if additional barriers are located between those two data manipulations; in effect, adjacent barriers coalesce into a single barrier.

Note that **sendData**s of a protocol suite may be grouped into integrated series differently from the corresponding **deliverData**s. Consider a multiplexing protocol for example. There is no problem in integrating **sendData**s across the multiplexing that occurs on the sending side, but it may not be possible to integrate **deliverData**s across the demultiplexing on the receiving side.

7.1 Control Transfer Barriers

Serial and integrated implementations represent different tradeoffs between memory access and control transfer. Serial implementations minimize the number of control transfers—just one per layer—at the expense of the many memory accesses necessary to pass a large amount of data all at once. Integration attempts to minimize memory accesses at the expense of frequent control transfers—one for every data word per layer. If control transfer is slow, the optimal choice is the purely serial implementation. For example, transferring control between protection domains (address spaces) is far slower than a function call, which is already much slower than the instruction sequencing used in our design. Hence, a serial implementation is more efficient than integration across protection boundaries. Another example is passing control between an outboard processor and the host processor, or between different processors of a multicomputer.

Although it is inefficient to integrate protocols across such boundaries, it can be advantageous to integrate *boundary crossing mechanisms* with protocols on one side of the boundary. For example, writing message data into a user address space can be integrated with protocols in the kernel address space. As another example, if a network device or outboard processor interface uses Programmed I/O, reading from or writing to such devices can be integrated with data manipulations on the host.

Clearly, it is desirable to minimize the number of control barriers that must be crossed. Ideally, all data manipulations would go in the same protection domain. Moreover, since certain protocols may have to be located in a particular protection domains—e.g., presentation formatting must often be performed in the application's address space—it makes sense to execute all the data manipulation protocols in that user address space.

7.2 Reassembly Barrier

From the perspective of higher layers, a message that is delivered to a reassembly protocol layer is not a message—it is only a fragment. A fragment does not even contain the higher level headers needed by higher level protocols unless the fragment is the first of a composite message. In general, protocols cannot process part of a message without having previously processed all the preceding parts of the message. Even if one could guarantee that fragments would always arrive in order and unduplicated, there would remain problems due to the possibly interleaved arrival of fragments of different messages. Higher level protocols would have to maintain state information for each composite message being processed, saving it between fragments and applying the corresponding state information to each fragment. It is not clear whether this can be accomplished while still supporting the efficient control transfers necessary for Protocol Integration. Gunningberg et al. [7] report that when they attempted to integrate fragmentation/reassembly, their implementation schemes lost the layer abstraction. A general technique for integrating across reassembly layers—one independent of the particular protocols—would be even more difficult to devise, if it is possible at all.

Fortunately, fragmentation/reassembly can be exiled to the bottom of the protocol graph, if it is needed at all. Gunningberg et al. [7] believe that "for multimedia applications and gigabit networks we will see fragmentation only at the lowest layers or not at all," and Clark and Tennenhouse's Application Level Framing [4] also places fragmentation/reassembly at the bottom of the protocol graph. Application Level Framing proposes that a single notion of message framing—an Application Data Unit—be used throughout the protocol graph. Only at the bottom of the protocol graph are Application Data Units fragmented into or reassembled from the Transmission Data Units supported by the network.

7.3 Random Access Barriers

Some protocols require random access to message data. A good example is the image transfer protocol described in [15]. This protocol's data manipulation reorders data so that pixels that are adjacent in the original data (an image) are spread far apart from each other as the data is transmitted over the network in packets. Thus, neither the sending nor receiving side of this protocol can efficiently manipulate data in a stream.

Another way of looking at random access barriers is that such protocols have a very large natural unit and cannot process anything smaller. The technique of accumulating input data in local state until a protocol's natural unit is available decreases in efficiency as the size of the natural unit increases. Clearly, for a natural unit on the order of the size of a message, a serial implementation would have better performance.

7.4 Buffering for Retransmission

A protocol that buffers copies of messages for possible retransmission presents a number of potential problems. First, consider retransmission. Retransmission involves the initiation of a data-containing message at a layer other than the top layer. An integrator could accommodate this by looking for invocations of **sendData**s in non-message processing operations at all protocol levels. It would then synthesize a series of integrated **sendData**s starting with the layer following the retransmission layer, in addition to any other integrated **sendData**s which have the retransmission layer in the middle.

Now consider buffering a copy of a message. Making a physical copy of a message, even if integrated, still entails writing the copy into the memory hierarchy. In a sequential implementation, it is often unnecessary to make a physical copy of a message since it is possible to retain or share an existing message data structure. In the integrated case, one would likewise prefer to make a logical copy whenever possible, i.e., whenever a data structure containing the correct version of the message data is available for some other reason. The only circumstances under which this does not hold are integrated series that include read-write data manipulations (in which category we include message data generators and consumers) both before and after the buffering layer. Otherwise, there is a message data structure (either the input or the output message) with the data in the correct form. These circumstances are determined at composition time, but that is too late—one needs to know at protocol design time because a single version of the buffering protocol code is not capable of supporting both logical and physical copying.

Thus, integrating efficiently across a retransmission layer complicates synthesis considerably. If we instead choose to treat retransmission as a barrier, it may be possible to simply avoid it. For example, it seems reasonable to locate it in the protocol graph so that it is either above or below all data manipulations. For example, end-to-end retransmission at the application level makes sense because only the application knows exactly its requirements, e.g., applications may chose to regenerate the message, or simply skip it, rather than buffer and retransmit.

7.5 Runtime Protocol Path Barriers

We have presented an implementation of integration in which all three stages of each protocol's **send** or **deliver** are combined in a single function. We refer to this implementation as "hard-wired" because for a given operation and a given first protocol, the set of protocols that are integrated with it is fixed. Hence, this technique can only integrate series of protocols which are known at integration time to be involved in processing any message received by the first in the series.

There are common cases where the protocols to be invoked subsequent to a given protocol are not fixed at compile time. For example, if a protocol multiplexes, then demultiplexing determines at runtime which protocol is to follow when receiving an incoming message. The sending side analog is the generalized routing exemplified in [14], in which the lower level protocols for an outgoing message are determined at runtime. Another example is message forwarding, since a message can either continue up the protocol graph if its destination is local, or be sent back down the protocol graph if it should be forwarded. Yet another example is flow control in which a message can be blocked temporarily; a message may either continue through more layers, or terminate, for the time being, at the flow control layer.

One technique that has been applied to a similar problem is the Packet Filter [12]. A packet filter is software that can be programmed at runtime to interpret the combined protocol headers on an arriving message to determine the message's eventual destination. This approach could be used to anticipate the path that a particular message will follow through the protocol graph, and select the corresponding integrated series of protocols. This technique does not address the generalized routing case or the flow control case, however. Furthermore, it adds redundant processing of message headers, and complicates protocols with the requirement to program the packet filter at runtime.

We have devised a technique called *lazy messages* to support integration in cases where the sequence of protocols to be traversed by a given message is not determined until runtime. Essentially, it allows initial stages to be executed before binding a message to a particular integrated data manipulation phase. Each protocol's initial stage is implemented as a separate function. The initial stage determines the subsequent layer. Each protocol's initial stage tags the message with the identity of that protocol, and attaches any information needed by the data manipulation or final stages of the same protocol. (There is a message header data structure to which information can be attached, even if the data portion of the message is yet to be generated.) When, in the sequential execution of initial stages, a layer representing a barrier is encountered, the protocol identifier tags accumulated on the message are used to select the function containing the corresponding integrated series of data manipulation and final stages. Such functions are constructed at compile time for each possible integrated series of protocols. The lazy messages technique is discussed in more detail elsewhere [1].

7.6 Discussion

The engineering *principle* (as opposed to the strategy) referred to as Integrated Layer Processing [4] states that a protocol should be "so structured as to permit the implementor the *option* of performing all the manipulation steps in one or two integrated processing loops" (our emphasis). For most of the barriers we have discussed, this option exists; they are not impossible to integrate, but an integrated implementation would be slower than a strictly layered implementation. Nonetheless, integrating data manipulations across such a layer could benefit overall performance enough to justify that particular layer being slower than it otherwise would be. An example of this is PES. Our measurements showed that integrating BSWAP, PES, and CKSUM data manipulations gave better performance than a serial implementation, even though the serial implementation used a faster, random access version of PES. The integrated implementation was faster because the gains due to eliminating loads and stores of data more than compensated for the slower, filter version of PES. In other words, the question of whether a protocol should be integrated is context-dependent; which protocols should be integrated and which should be strictly layered depends on the relative order of data manipulations and barriers.

This is not a problem if the focus is implementing a particular protocol graph. In this situation, the composition of the protocol graph is known at protocol implementation time, and therefore, one can take into account whether or not it will be integrated. The issue becomes a problem when reusability of the protocols is a goal. The problem is that neither a buffer-layered or a stream-filter implementation of the protocol is optimal for all the contexts in which it might be composed. One response to this problem is to maintain two versions of such protocols, a stream-filter version to use when integration wins, and a buffer-layered version to use when strict layering wins. If lazy message integration were used, one could conceivably benefit by using both implementations for a single protocol instance, since the optimal implementation could depend on which particular branch is taken by a message.

8 Concluding Remarks

Protocol integration is an approach to implementing protocols that results in higher throughput than serial implementations by increasing the locality of message data access. The protocol integration design presented in this paper supports a broad class of protocols, requiring only that they maintain the invariant of an integral number of message data words. Furthermore, this design is largely independent of machine architecture and compiler.

The performance of integrated protocols compared to serial implementations depends on the relative speeds of computation, cache access, and memory access. Protocol integration will increase its advantage over serial implementations if, as expected, the cost of accessing a cache-mediated memory hierarchy increases relative to computation. To quantify the benefit of integration, we studied several example protocols on three different machines. At one end of the spectrum, on the HP 720—of the machines we measured, the one with the most pronounced processor-to-memory performance discrepancy—integrating seven layers improved throughput by a factor of two to nearly four times, depending on cache effects. At the other end of the spectrum, even integrating just two simple layers improved throughput 20 – 35% over the three machines we measured.

Automatic integration is the synthesis of an integrated series of protocols from modular implementations of the component protocols. Custom integration sacrifices modularity in return for performance, but automatic integration achieves high performance while retaining modularity. Furthermore, automatic integration supports reuse of protocol implementations, since a protocol module may be automatically integrated into various protocol graphs with no additional implementation effort.

Acknowledgements

Peter Druschel provided many insightful comments about this work, including the observation that integration restructures a computation with poor locality of data reference into one with good locality. In addition, Hilarie Orman, Richard Schroeppel, Michael Pagels, and Todd Proebsting provided helpful feedback.

References

[1] M. B. Abbott. *A Protocol Architecture and Implementation Language*. PhD thesis, University of Arizona, August 1993. In preparation.

[2] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1), Feb. 1993. To appear.

[3] 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.

[4] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept. 1990.

[5] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 91.

[6] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Analysis of I/O subsystem design in multimedia workstations. In *Third International Workshop on Network and Operating System Support for Digital Audio and Visual*, pages 251–263, San Diego, Calif., Nov. 1992.

[7] P. Gunningberg, C. Partridge, T. Sirotkin, and B. Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the 2nd MultiG Workshop*, 1991.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[9] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas. Tools for implementing network protocols. *Software—Practice and Experience*, 19(9):895–916, Sept. 1989.

[10] N. C. Hutchinson, L. L. Peterson, M. Abbott, and S. O'Malley. RPC in the *x*-Kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 91–101, Dec. 1989.

[11] B. W. Kernighan and D. M. Ritchie. The m4 macro processor. In *Unix Programmer's Supplementary Documents Volume 1*. University of California at Berkeley, Apr. 1986.

[12] 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 System Principles*, pages 39–51, Nov. 1987.

[13] NBS. Data encryption standard. Technical Report Federal Information Processing Standards (FIPS), Publication 46, National Bureau of Standards, US Department of Commerce, Jan. 1977.

[14] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[15] C. J. Turner and L. L. Peterson. Image transfer: An end-to-end design. In *Proceedings of the SIGCOMM '92 Symposium*, Baltimore, Maryland, Aug. 1992.