Speeding up Protocols for Small Messages

Trevor Blackwell
Harvard University
29 Oxford St
Cambridge MA 02138
tlb@eecs.harvard.edu

Abstract

Many techniques have been discovered to improve performance of bulk data transfer protocols which use large messages. This paper describes a technique that improves protocol performance for protocols that use small messages, such as signalling protocols, by reducing memory system penalties. Detailed measurements show that for TCP, most memory system costs are due to poor locality in the protocol code itself, rather than movement of data. We present a new technique, analogous to blocked matrix multiplication, for scheduling layer processing to reduce memory system costs, and analyze its performance in a synthetic environment.

1 Introduction

As widely available network speeds have increased by more than an order of magnitude, signalling protocol performance has not increased nearly as much. Perhaps due to the ease of benchmarking bulk data transfer protocols (it is easy to compare Mbits/sec,) considerably more attention has been paid to data transfer than to signalling performance.

The main way to improve the performance of data transport protocols is to reduce the number of operations on the data. Traversing large blocks of data is especially time consuming on modern processors because when the data does not fit in the on-chip cache, the processor must wait for slower external memory.

But data and signalling protocols have very different characteristics. We think of signalling protocols in the broadest sense of protocols whose ultimate purpose is not data transfer. Such protocols are ubiquitous in the Internet: DNS, ICMP, IGMP, TCP's connection control messages, all except two messages in NFS, to name just a few.

We are especially interested in the performance of Q.93B, the standard ATM connection setup protocol. Whatever one's views on the proper place for ATM, it is clear that it would be useful in more environments if VCs could be set up as cheaply as, say, TCP connections. A major obstacle is processing time in the switches. If ATM switches are deployed like IP routers, than a cross-country connection might pass through 10 to 20 switches. Several current signalling implementations spend 5 to 20 milliseconds processing each message: this could add a large fraction of a second to the connection setup time across a large network, and limit the aggregate rate of connection setups through any switch to 50 to 200 per second. Our performance goal is to support 10000 pairs of setup/teardown requests per second with processing latency of 100 microseconds for setup requests, using just a commodity workstation processor.

In most cases, all the signalling protocols mentioned above send small message, on the order of a hundred bytes or less. For these protocols, the cost of reading and writing messages is not a major bottleneck.

This paper presents measurements to show that even for TCP, a fairly simple data transport protocol, the volume of code touched per packet can be much larger than common cache sizes, and that the processor spends more time loading protocol code from memory than moving packet contents. Measurement presented in Section 2 show that in the common case of bulk data transfer over an internetwork, a common TCP implementation running on a common RISC processor spends ten times longer fetching protocol code from memory than moving message contents to and from memory.

Appears in ACM SIGCOMM '96.

Sometimes, computations can be restructured to improve the locality of reference. For instance, matrix operations can be "blocked" [23] by rearranging loops to achieve dramatic performance increases. We propose a new way of scheduling protocol layer processing for small-message protocols which, like matrix blocking, can dramatically reduce memory system penalties under heavy loads.

In this paper, we concentrate on receive-side processing to simplify the discussion. The techniques presented are also applicable to transmit-side processing, but we have not evaluated their performance or considered implementation in detail for the transmit case.

The rest of Section 1 gives some background on the impact of memory locality on system performance. Then section 2 describes detailed measurements of TCP/IP which demonstrate the large working sets of even simple protocols. Section 3 describes the central idea of this paper: locality-driven layer processing (LDLP). Section 4 analyzes the performance of LDLP in a synthetic system. Section 5 describes other ways to improve locality in protocol stacks, and section 6 concludes.

1.1 Performance of Data Transport Protocols

A naive layered protocol implementation might copy message payloads at every layer. But for a long time, protocol implementors have utilized buffer management schemes that allow common operations such as stripping headers and concatenating fragments to be done without copying any message contents. The `mbuf` system of 4.2BSD [2] is perhaps the canonical example; the x-kernel [18] supports a more powerful scheme.

Much research has focussed on integrating repetitive operations on message data. Integrated layer processing (ILP) [7] tries to reduce data movement by loading each byte of a message once, and performing the operations for multiple layers on it, rather than reloading and possibly storing the message contents for each layer.

The principle of Application Level Framing [7] encourages applications to transfer data in units that can remain intact through many layers of the protocol stack. It reduces the amount of queuing that must occur between layers, allows each message to stay in the data cache during its entire trip through the protocol stack, and facilitates implementation of ILP.

For protocols that do simple processing on large messages, moving message contents into and out of the processor is usually a major bottleneck, and thus techniques to reduce data movement have been shown to give significant performance benefits when protocols use message sizes of a few kilobytes or more. Druschel's thesis [16] gives a good description of ILP, copy elimination, ALF, and other techniques to reduce data movement costs.

1.2 Memory System Performance

Modern RISC CPUs achieve high performance with the help of on-chip caches which, for many workloads, can satisfy a large fraction of data and instruction references without incurring the delay inherent in accessing memory separate from the CPU. Software performance depends on both the number of instruction execution cycles and the number of cycles spent waiting for memory.

Ousterhout [11] showed in 1990 that largely because of memory system stalls due to poor locality of reference, kernel code had benefited little from several years of processor performance improvement which made an order of magnitude improvement in a large class of other applications. Architectural decisions have been shown to have a large impact on system performance because of memory system effects, even when the number of instructions executed does not change significantly [13, 12]. It is clear that locality needs to be treated as an important and fundamental property of system designs.

Currently, several systems with fast CPU clock rates, small primary caches of 8 or 16 KB, and substantial CPU/memory bandwidth mismatches have good overall price/performance ratios. For example, the DECStation 3000/400 [24] comes with an 8KB primary instruction cache, and wastes 20 instruction slots (10 cycles) due to a primary instruction cache miss that hits in the secondary cache. This processor uses 32-byte cache lines, which means that data is fetched from memory in units of 32 bytes. Rosenblum [19] predicts that some high-performance processors will have 64 KB caches by 1998, but that the number of instruction slots wasted due to a primary instruction cache miss will climb to 60.

All the results presented here are measured or simulated for the DEC 3000/400. We think the principles apply to any machine with small primary caches.

2 Measurements of TCP/IP

Much work has already been done on optimizing the BSD-derived TCP/IP system [2], and it is not particularly a goal of this paper to suggest ways to improve it. Rather, measurements are given for the implementation of TCP in 4.4BSD because although it represents a mature and well-tuned implementation of a fairly lightweight protocol, the working set sizes are surprisingly large. One might expect that most memory system costs would be due to movement of the data being transported. However, our measurements show that message contents count for less than 10% of the memory system traffic for the common case of large data transfers over an internetwork.

To aid in the discussion of locality in protocol implementations, we define a few terms:

Data Loop. The loop performed at one protocol layer (or once for multiple layers with ILP) iterating over the message contents.

Working Set. The subset of code and read-only data for the entire protocol stack that is referenced in the receive path. Code and read-only data are deliberately merged because they contribute similarly to memory bandwidth usage, and because programmers and compilers are often free to trade between the two.

Measurements were taken from a trace of a TCP socket receiving a message, delivering the contents to the application, and sending an acknowledgment as detailed in Table 2. Note that this TCP implementation sends an ACK for every second data packet; thus the measurements presented are for the larger of two common cases of receiving a packet. Figure 1 shows a map of the working set.

2.1 System Description

We measured the TCP implementation of NetBSD/Alpha (based on 4.4BSD Lite) as of September 1995, with locally optimized versions of a few routines, configured for minimal error checking in the kernel, and compiled with full optimization. It runs on a DEC 3000/400 (“Alpha”), using the Lance Ethernet driver. TCP’s timestamp-related features [1] are not enabled.

We think the working set size of this implementation is typical of BSD-derived TCP/IP stacks on RISC workstations. Common causes of large code size, such as function inlining, are not done. Primary cache lines are 32 bytes: a reference to any element in the cache line makes the whole cache line part of the working set. The large cache line size does affect the working set sizes, especially for read-only data which tends to be sparse. With 16-byte cache lines, the working sets for code and read-only data would be 13% and 31% smaller. Section 5.3 discusses the effect of cache line sizes in detail.

The 64-bit register size of the Alpha does not increase the working set sizes very much over a 32-bit machine. Although many data structures are larger for 64-bit machines, few such data structures are counted in the read-only data. Individual 64-bit pointers are common; however these require exactly one cache line on either a 32- or 64-bit machine. Long code sequences to generate 64-bit constants do not occur often enough to contribute significantly to the code size.

2.2 Tracing Apparatus

Tracing was done under NetBSD by simulating instructions and recording all memory references to a trace buffer for later analysis. A special function `alphasim_entry` was added to the kernel to invoke the tracing apparatus. Rather than returning like a normal function, the simulator creates a new temporary stack for its own use, and starts simulating instructions starting from the instruction after the call to `alphasim_entry`. All memory references are logged to a trace buffer, which can be read by a user process. Calls were added at the system call, context switch, and hardware interrupt entry points. A flag controls whether or not the tracing system is enabled. The kernel runs normally when tracing is disabled, and runs about 20 times slower when it is enabled. In some ways, this system is more convenient than code modification systems such as ATOM [15], which cause the traceable code to run slowly whether or not traces are actually being collected.

The tracing system can handle nearly all Alpha instructions. When it encounters an instruction that it can’t handle, it stops tracing and jumps back to the original code. The only such instructions occurring in the traces were the special instructions which return from an interrupt or system call; these were logical endpoints for the individual traces.

The simulator was validated by linking it into the compiling pass of the GCC compiler and compiling several large programs, and also by running a variety of programs on an instrumented kernel with tracing enabled. Except for the slowdown, no differences were noted.

Several programs were used to combine and analyze the individual traces. We experimented with several formats for visualizing the traces in addition to the format used for Figure 1. As the tracing system can also produce a procedure call graph, it has also been generally useful in understanding control flow in the kernel

One reason for choosing the Alpha processor for the experiments was the relative ease of writing an instruction set simulator. The whole instruction simulation and tracing apparatus was less than 1000 lines of C and assembler.

To validate the results, similar measurements were made using an entirely different system. The ATOM code modification system [15] was used to modify the Digital Unix 3.0 kernel to record a trace of its memory references. Results were similar: Digital Unix’s working set had about 30% less code, and a similar amount of data. The smaller code size is partly due to some linker optimizations that simplify nonlocal procedure calls, and partly because Digital's implementation is better tuned for the Alpha. Both these sets of results miss some contributions to the working set due to PAL code, an Alpha-specific architectural feature which implements certain low-level functions such as switching between user and system mode, and managing the translation lookaside buffer (TLB). We do not currently have access to the PAL code.

2.3 Trace Collection

Measurements were done with a program which:

- opened a socket to a process on another machine which wrote data continuously
- enabled tracing
- received several TCP segments
- disabled tracing

The trace buffer was then dumped to a file and analyzed.

2.4 Results

Description		**Working Set Sizes**		
		Code	**Read-only Data**	**Mutable Data**
Protocol Layers	Device	4480	864	672
	Ethernet	2784	480	128
	IP	3168	448	160
	TCP	5536	544	448
	Socket low	608	32	160
	Socket high	1184	256	64
Over-head	Kernel entry/exit	2208	1280	640
	Process control	5472	544	736
Common	Buffer mgmt	1632	192	512
	Copy, checksum	3232	448	128
Total		**30592**	**5088**	**3648**

TABLE 1. **Breakdown of Working Set Sizes in NetBSD TCP Receive & Acknowledge Path.** No accesses to packet contents, mutable data structures, hardware registers, or the stack are counted. Unit of memory is a 32-byte cache line. Data is considered read-only if it was not modified during the trace. Code is classified into layers based on its function; data is classified based on the function executing when it was first accessed during the trace.

Table 1 shows the breakdown of working set contributions for the various parts of the protocol stack, based on a detailed classification of memory references. In total, about 30 KB of code and 5 KB of read-only data is

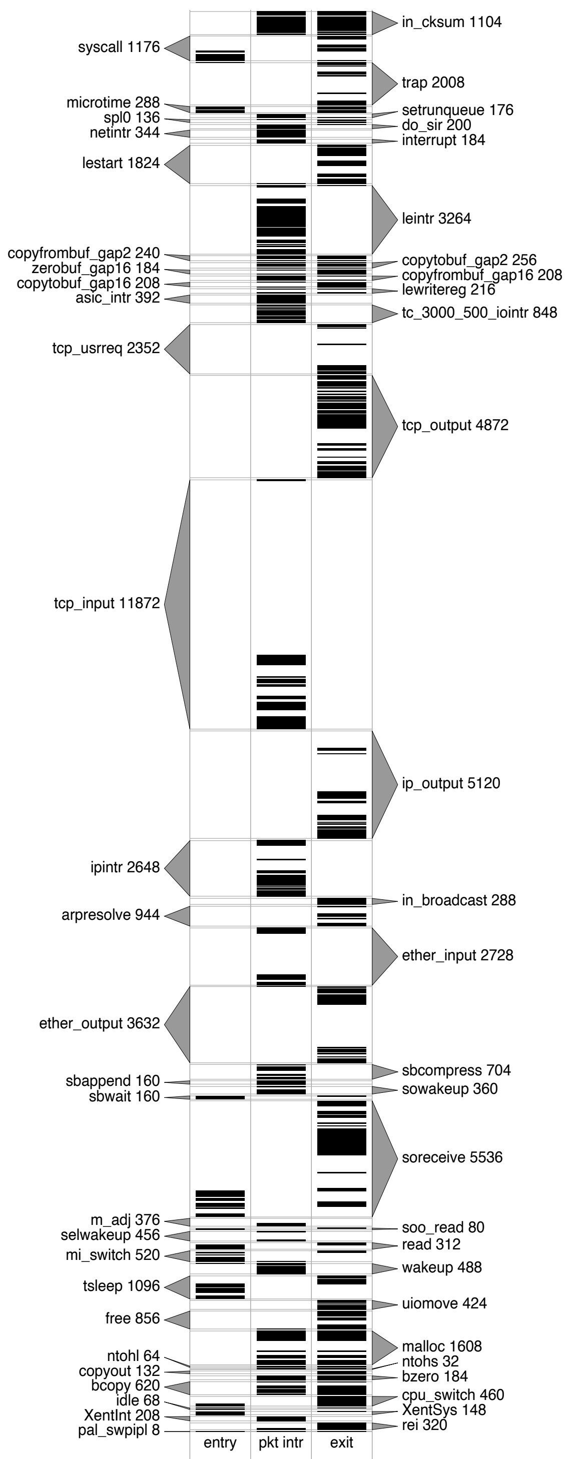

FIGURE 1. **Plot of Active Code for TCP Receive & Acknowledge Path.** Vertical axis is memory address in code segment, divided into functions. The three columns are the phases of the trace: process making read call and being blocked; device interrupt and adding payload to socket buffer; process waking up, copying data into process memory, and sending ACK. Numbers beside function names give the total size of the function in bytes; only the bytes actually touched are counted in Table 1.

Entry	Process makes read system call. Call is dispatched to socket layer. No data is available in socket receive buffer, so process sleeps.
Device Interrupt	Message arrives on Ethernet, and triggers device interrupt. An mbuf is allocated, the message is copied from device memory into the mbufs, and the mbuf is placed on a received message queue. As soon as the interrupt returns, further processing happens at a lower interrupt level. The message is vectored through the IP layer, which does very little because the message is addressed to the host and is not a fragment, and then to TCP. TCP is able to use its fastpath, and the single-entry PCB cache hits. It computes the checksum for the message, updates sequence number and timer fields in the PCB, and delivers the contents to the socket layer. The socket layer appends the data to the socket receive buffer and wakes up the sleeping process.
Exit	The process wakes up. The socket layer checks the receive buffer, finds data, and copies it into the process's address space. It calls the TCP layer to send an ACK, and returns from the system call.

TABLE 2. **Phases of TCP receive & acknowledge path shown in Figure 1.**

touched every time a packet is received and an ACK generated.

The unit of granularity for all memory references is a cache line. Code is classified into layers based on its function. Common functions used by multiple layers, such as buffer management, data copying, and checksum routines, are classified separately. The classification of data into layers is somewhat approximate. Cache lines which contain data relevant to multiple layers are assigned to whichever layer referenced them first during the traces.

Clark et al [6] reported measurements of TCP/IP which showed that the overhead of receiving a message and sending an acknowledgment (i.e. all but the per-byte costs) was 639 i386 instructions. Assuming the same number of Alpha instructions would suffice, 639 distinct instructions on the Alpha requires only 2556 bytes of code. How can this number be reconciled with the results above? Clark eliminated many elements which are included here. His analysis does not include the socket layer, interrupt dispatching, system call overhead, process management, memory allocation, or the device driver. He used a stripped down buffer management system that would be inadequate for a general-purpose TCP implementation. His analysis eliminates not only the per-byte costs of the checksum and copying routines, but also the fixed overheads of these routines. Thus Clark's numbers measure code that implements only a small fraction of the functionality of the 4.4BSD protocol stack.

On machines with 8 KB caches, the hit rate of TCP will not be good, as the working set is more than four times larger than the cache. Few lines will remain in the cache between successive iterations of the receive & acknowledge path. Assuming a good cache layout such that conflicts do not cause cache lines to be loaded more than once per iteration, about 35 KB of code and read-only data is fetched and discarded from off the CPU. The message contents (between 512 and 584 bytes depending on the layer) are fetched twice into the primary cache and stored twice for an off-CPU IO volume of 2.2 KB in most cases. Thus it is clear that message contents are not the main consumer of precious memory bandwidth.

3 Restructuring for Locality

Figures 2 and 3 show the fundamental idea of restructuring protocol stacks to improve locality. Figure 3 represents protocol layers as rows in a left-hand matrix, and messages as columns in a right-hand matrix. Protocol processing is analogous to matrix operations such as multiplication, because each layer of the protocol stack must be applied to each packet. Figure 2 shows the same concepts in a schematic loop structure.

For useful work to occur, both protocol code and message contents must be brought together in the processor. Conventional protocol implementations generally take one message at a time, and execute the code for all layers it must pass through in turn, as embodied in the ALF principle. Integrated layer processing modifies the conventional stack so that multiple layers are applied to each message simultaneously, as shown in the middle column in Figure 2.

While processing one message at a time works well when the working set of all the protocol code fits in the cache, it results it poor performance when it does not. For protocols that have small messages and large layer implementations, it is better to bring the code for a single layer into the cache, and apply it to several messages. This can be visualized as bringing the (small) data to the (large) code, rather than the reverse.

Blocked layer processing generalizes the notion of optimizing locality in layer processing. Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies when performing matrix operations. Rather than looping over all rows and all columns at once, subsets of the rows and columns are taken, and the innermost loops are confined to these subsets. The result is increased locality. In the conventional structure any given layer has been evicted from the cache by the time it is accessed again. In the blocked structure, layers are used multiple times before being evicted.

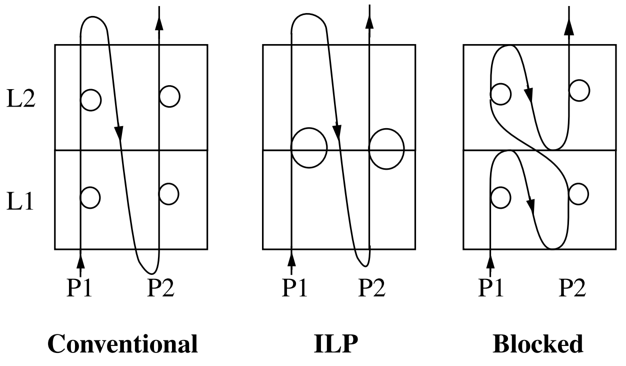

FIGURE 2. **Schematic Loop Structure of Conventional, ILP, and Blocked Protocol Implementations.** Only two layers (L1, L2) and two messages (P1, P2) are shown, but the principle applies to many layers and messages. For each implementation style, the control flow to receive two messages is shown. Small circles are loops over the data. **Conventional**: Each message is processed by each layer in turn. Outer loop has poor locality. **ILP**: Each message is processed by each layer in turn, but loops over data are integrated. Outer loop has poor locality. For protocols that do complex data processing, such as encryption, inner loops may also have poor locality. **Blocked**: Multiple messages are processed by each layer in turn. All loops have good locality. Because more messages are being processed simultaneously, locality is reduced for message contents.

3.1 Locality-Driven Layer Processing

Blocked layer processing is an off-line algorithm — that is, it assumes a preexisting sequence of packets. Protocol stacks need to be on-line, responding to messages when they arrive.

A simple way to implement blocked layer processing would be to wait until, say, 5 messages had arrived, and then process them using a blocked loop structure. Clearly this is unacceptable for real protocol stacks, as any message might have to wait an unbounded amount of time before a complete block arrives.

A better way is to process batches consisting of all *available* message. Assume for simplicity that when messages arrive, they are buffered in the adaptor hardware. When the protocol stack is able to accept a new message, it takes all available messages and processes them in a blocked pattern. When it is finished, it again looks for new messages.

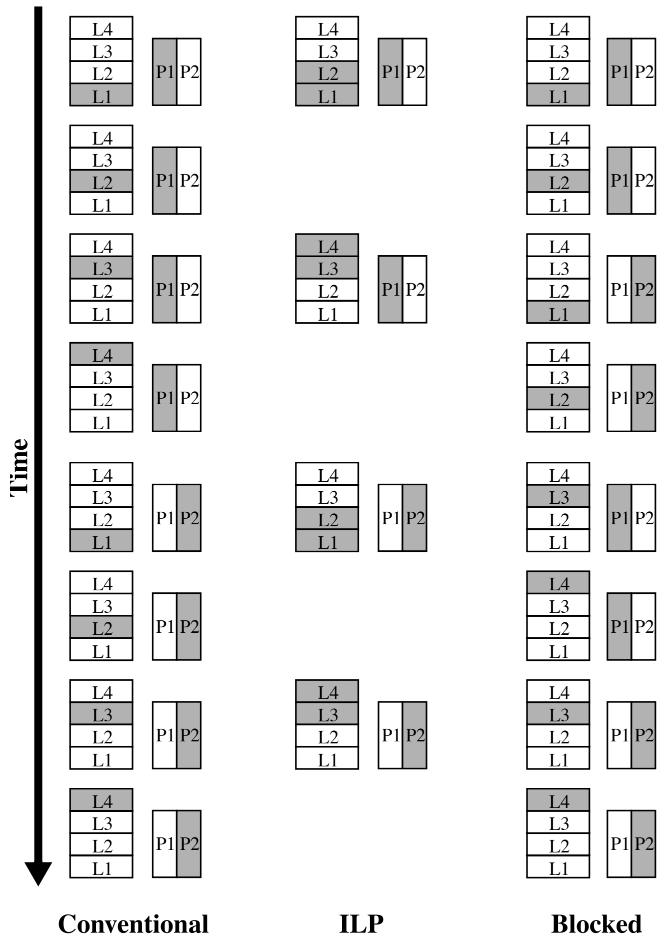

FIGURE 3. **Layered Procotol Stacks as Matrices**. Protocol layers (L1, ... L4) are rows in the left matrix; packets (P1, P2) are columns in the right matrix. The sequence of computation is shown down the page. **Conventional:** Each layer is applied to each message in turn. **ILP:** Multiple layers are applied to each message in turn. **LDLP:** A subset of layers and messages that will fit in the cache is chosen, and and each layer in the subset is applied to each message in the subset in turn.

Under light load, messages will usually be processed singly, minimizing delay. Under heavy load, messages will be processed in batches, maximizing throughput. This algorithm is simple to implement, and the results in the next section show that it works well — increasing throughput and decreasing latency at most load levels.

One can roughly categorize protocols as handling large messages or small messages, according to the relative sizes of typical messages and protocol code. Figure 4 makes this distinction slightly more concrete. Large-message protocols are those where the size of most messages is larger than the size of the protocol code which is normally referenced for each message, both measured in terms of the memory they occupy; small message protocols are the opposite. For large-

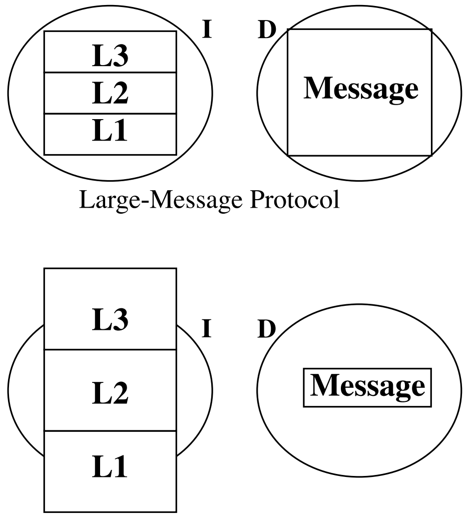

FIGURE 4. **Large- and Small- Message Protocols.** For both types, protocol layers are shown superimposed on the processor's primary instruction (I) and data (D) caches. For large-message protocols, the size of a message is larger than the working set of the protocol itself; for small-message protocols the working set is larger than the message. Although separate instruction and data caches are depicted, the results of the paper hold equally well for processors with unified caches.

message protocols, one is a good blocking factor, and so a conventional protocol implementation performs well. It is small-message protocols which benefit from LDLP.

3.2 Implementation

The optimal blocking factor is hard to estimate. Lam [22] presents algorithms that can give a fairly accurate estimate of the optimal blocking factor.

For many signalling protocols, just one layer will fit in the instruction cache, while several messages fit in the data cache. For this special case, implementation is especially simple. Messages are processed in batches consisting of as many available messages as will fit in the data cache.

In protocol stack implementations where there is a queue between adjacent layers and each layer is implemented as a separate task, implementing LDLP is a simple matter of task scheduling. Higher layers are given higher priorities, but all layers run to completion — that is, they process all the messages in their input queue. The lowest layer, however, is made to yield the CPU after processing as many messages as will fit in the data cache.

For protocol stacks which pass messages between layers with just a procedure call and no intervening queue, the entry point to each layer is modified to append the message to a queue of messages to be processed for that layer, and then return. When a layer is invoked, it pulls messages off its queue, making calls as usual to the next layer to propagate messages upward, until the queue is exhausted. Then, it invokes all layers that can be directly above it (there can be more than one) to process the messages in their queues. Some informal experiments suggest that enqueuing and dequeuing messages costs on the order of 40 instructions.

This scheme requires a buffer management scheme where lower layers hand off their buffers to the higher layers, and don't destroy them after calling the upper layers. The 4.4BSD `mbuf` system works well.

4 LDLP Performance Results

This section presents a synthetic benchmark which shows how LDLP can increase throughput by increasing locality. As batching can have the effect of delaying packets, we present mainly latency results. The results show that while latency increases slightly for a small range of input loads, the increased throughput decreases latency by reducing queuing at most load levels.

The synthetic benchmark simulates receiving messages through a five-layer protocol stack. Although few stacks implement five layers of the OSI reference model, "layer" need not directly correspond to an OSI layer. For instance, six candidate layers are shown in Figure 1.

The synthetic benchmark simulates a good cache layout for each individual layer. That is, within each layer, there are no self-conflicts. Such a good layout is probably feasible with commonly available tools such as Cord [14]. Large numbers of cache conflicts within a layer would tend to reduce the relative benefit of LDLP along with overall performance.

Measurements are shown for 8 KB direct mapped instruction and data caches. Because the caches are not fully associative, the number of conflict misses depends on the way the program is laid out in memory. To insulate the results from the vagaries of layout effects, average results are presented from 100 runs, each with a different random placement in memory. Since each run lasted one second, the results for each arrival rate cover 100 seconds of simulated time.

The processor runs at 100 MHz, and every read cache miss causes it to stall for 20 cycles. This miss penalty, higher than that of the DEC 3000/400 used for results in Section 2, is used to emphasize the effects of locality. Some processors can prefetch instructions from the second level cache to hide some of the cache miss cost, although ultimately the execution rate is bounded by the second level cache bandwidth, and possibly by the main memory bandwidth for very large protocol working sets.

Each layer has 6 KB of code, and 256 bytes of data in its working set. Each instruction in the working set is executed at least once, including a 40-instruction loop over the data with a cost of 0.5 cycles/byte. In total 1652 cycles of instruction processing are executed for each layer. This benchmark somewhat overemphasizes the importance of touching data, as most protocol stacks to not touch all the data at every layer.

The input to the simulator is a stream of 552-byte messages (a common packet size in IP internetworks) from a Poisson traffic source. Figure 5 shows how the number of cache misses per message decreases at high arrival rates with LDLP, due to blocking. Figure 6 shows that latency is improved at almost all arrival rates. In Figure 5, it can be seen that data cache misses increase at high arrival rates and batching factors, but that the increased data misses are far outweighed by the decreased instruction misses.

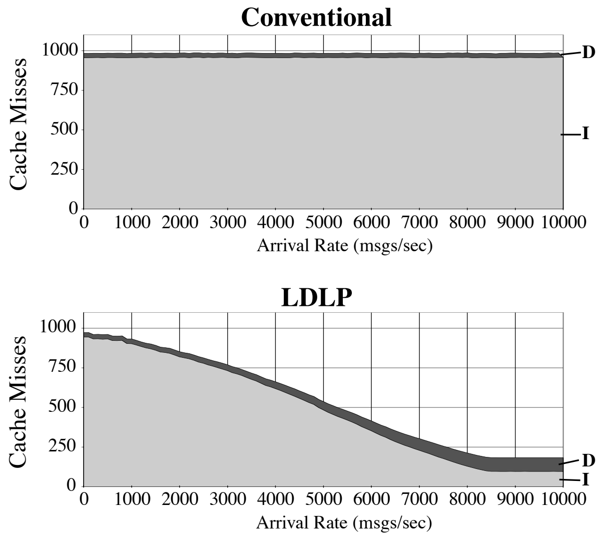

FIGURE 5. Cache Misses (Instruction and Data) per Message as a function of Arrival Rate using a Poisson traffic source. At high arrival rates, batching increases data cache misses, but decreases instruction cache misses. Because the code size is larger than the message size, the effect of decreased instruction cache misses dominates. The LDLP curve flattens out beyond 8500 msgs/sec because the level of batching becomes limited by the maximum batch size.

Because Poisson processes are not representative of many real-world traffic sources [20], we also used the traces of Ethernet traffic collected by Leland et al. [21] to drive the simulation. The first 1000 seconds of the October 5, 1989 trace were used. The packet sizes are taken from the traces, but all other parameters are the same as for the Poisson traffic source. Rather than varying the arrival rate (which is

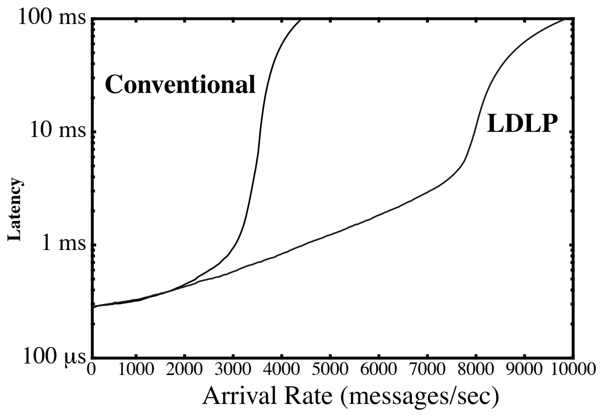

FIGURE 6. Latency as a function of Arrival Rate using a Poisson traffic source. Although batching can increase latency, little batching occurs at low arrival rates, and improved throughput reduces queuing at high arrival rates. Buffering is limited to 500 packets, so latencies much beyond 100 ms would involve packet drops.

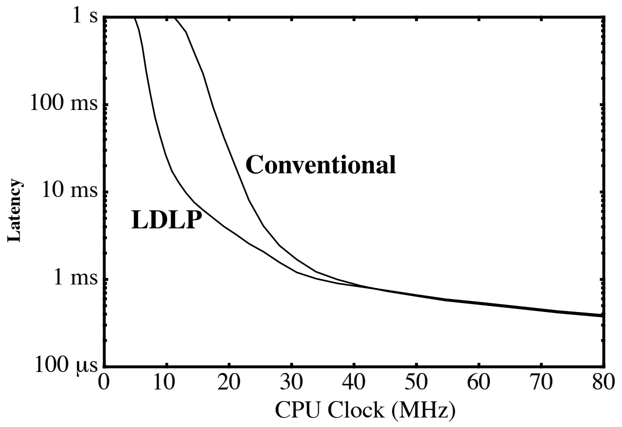

FIGURE 7. Latency as a function of CPU Speed using Ethernet traffic traces. In general, as CPU speed falls, latency increases. When processor speed falls below 40 MHz, the LDLP version batches packets to maintain throughput.

fixed in the traces) we varied the simulated CPU clock speed and observed the changes in latency. Figure 7 shows how latency is higher at lower clock speeds, but how LDLP increases performance under heavy loads.

5 Other techniques to Improve Locality

LDLP is mostly independent from the implementations of the layers themselves. That is, unlike ILP or copy-elimination schemes, it can be applied to existing protocol implementations by changing only the interface to the layers, while the code for the layers themselves should change very little. This section describes techniques to improve code locality by changing the implementation of the layers themselves.

5.1 Programming

It is common practice to unroll loops to minimize the per-byte costs of operations such as the TCP/IP checksum. The set of such optimizations is generally chosen experimentally by measuring performance over some distribution of packet sizes. Designers should heed the conclusions of section 2, and compare performance of alternative designs with cold, rather than warm primary caches.

For example, the elaborate `in_cksum` routine from 4.4BSD compiled for the Alpha weighs in at 1104 bytes, 992 of which are in the working code set for messages larger than 32 bytes. In the Digital Unix implementation, the working code set is about 700 bytes. We compared the performance of the 4.4BSD routine to a very simple version (288 bytes of active code) which was smaller, but required more processing per byte. With a warm cache, the elaborate version performed better at nearly all message sizes. With a cold primary instruction cache, the simple version had to fetch fewer instructions from memory, which made it faster for messages sizes up to 900 bytes. Measurements were made using the processor cycle counter on a DECStation 3000/400. The data being checksummed was in the cache in all cases.

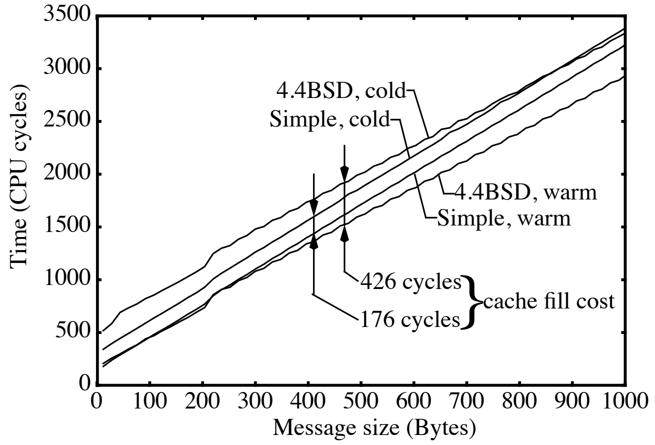

FIGURE 8. Cache Effects in Checksum Routines. The elaborate `in_cksum` routine in 4.4BSD achieves excellent performance for both large and small messages with a warm instruction cache. With a cold cache, however, the simple version's smaller code size makes it faster for messages sizes up to 900 bytes. To smooth out the curve, times for each range $[x \ldots x + 15]$ of messages sizes x are averaged together.

The 4.4BSD checksum routine was not tuned with the Alpha (or any particular processor) in mind, so these results do not reflect badly on its design. However, the result holds that any checksum routine which touches more than a few hundred bytes will be slow for small messages. Simple checksum routines, containing less than a few hundred bytes of code, are likely to be the best design choices.

A large fraction of the complexity of the checksum code is due to the buffer system. Clark commented [6] that "a buffer layer can easily grow in complexity to swamp the protocol itself," and our observations agree. In addition to the buffer management code shown explicitly in Table 1, a substantial fraction of all layer processing code is involved with peeking inside buffers, handling buffer discontinuities in the middle of protocol fields, ensuring that buffer overrun does not occur, and other things that could be omitted if large contiguous buffers could be used everywhere.

5.2 CPU Architecture

To launch one more volley into the CISC/RISC debate, it is interesting to note that CISC processors may have an advantage over RISC processors in protocol implementation due to increased code density (and may therefore benefit less from LDLP). Networking code is substantially smaller on the i386 than on the Alpha for at least the following reasons:

- General code is significantly smaller. The NetBSD TCP and IP code (all source files beginning with `tcp_` or `ip_`) is 55% smaller on the i386. Informal measurements of several other programs showed that typical i386 code was about 40% smaller.
- The i386 has special instructions for copying data — thus for example the `bcopy` function touches 64 bytes of code on an i386 with 32-byte cache lines, but 448 bytes on the Alpha for most common inputs.
- The i386 supports unaligned memory operations, which means that code to extract values from received messages does not need to handle misalignment.

Thus, with equal size caches, many medium-sized loops will have good locality on CISC machines such as the i386, but poor locality on RISC machines such as the Alpha.

5.3 Cache Organization

Large instruction cache line sizes are probably appropriate for protocol code. Table 3 shows how the number of bytes and cache lines in the working set changes for different cache line sizes. The numbers are derived from processing the TCP/IP traces using different machine models. Doubling the instruction cache line size to 64 bytes would, in the absence of conflicts, decrease the number of cache misses by 41%.

Cache Line Size	ΔSize in Bytes ΔSize in Lines Code	Read-only Data	Mutable Data
64	+17% -41%	+44% -28%	+55% -22%
32	0	0	0
16	-13% +73%	-31% +38%	-38% +23%
8	-20% +216%	-55% +81%	-56% +75%
4	-25% +500%	N/A	N/A

TABLE 3. **Effect of Cache Line Size on Working Set** for TCP/IP traces. 4, 8, 16, and 64 byte cache lines are compared to the baseline 32-byte cache line. For each entry, the percentage change in number of bytes and number of cache lines is given. Because the word size on the Alpha is 64 bits it would be infeasible to make the data cache lines smaller than 8 bytes, and these entries are marked as N/A

5.4 Compiler Optimizations

Mosberger [17] describes the application of compiler techniques to compact the working set of protocol code by moving rarely executed basic blocks to the end of functions to avoid diluting the cache with instructions that do not get executed. He shows a significant reduction in instruction cache misses, and an increase in overall performance.

The results above show that cache dilution in the TCP/IP traces is significant. For example, one can conclude from the numbers in Table 3 that about 25% of instructions fetched into the cache are not executed, and therefore that a perfectly dense cache layout would reduce the number of cache lines in the working set by about 25%. Instruction prefetching increases the relative benefit of dense cache layouts.

6 Conclusions

Before following the prevailing theory that ILP, ALF, and copy avoidance schemes constitute the road to high performance, designers should decide whether their protocol will process mostly large or small messages. In comparing message size to protocol working set size, It is important to include all overheads of dispatching at layer boundaries, process and thread management, and memory management in predicting the size of the protocol, as these can be surprisingly large. The complete flow through a protocol should be considered, including such functions as sending acknowledgments.

Developers who want their small-message protocols to run fast must understand the locality characteristics of their code. A reasonable procedure when implementing protocol stacks from scratch is to write layers as independent units, measure their working sets, and then decide how to group them to maximize locality.

For nontrivial protocols that do not use LDLP running on workstations with small primary caches, designers should assume, only slightly conservatively, that every message received causes every piece of code executed for that message to be fetched into the primary cache at least once. That is, at the start of processing each message, the cache is cold. Any additional code added to speed up processing incurs memory system costs - at least one extra cache miss (costing, say, 10 cycles) for every extra cache line (say, 32 bytes).

LDLP can be used to make small-message protocols go faster without significant change to the implementation of individual layers. LDLP is most compatible with the more straightforward ways of implementing layers: disentangling layers written for ILP is not a job for the faint of heart.

It was a surprise to us that LDLP could be advantageous with protocols such as TCP that are simpler than the signalling protocols we are targeting. In particular, LDLP may improve performance for Internet WWW servers, where the data transfer unit is 512 bytes or less in most circumstances.

If the future brings processors with large primary caches, will LDLP become irrelevant? It depends on the protocol. The working set of most efficient data transport protocols will probably fit in 64 KB caches, which may be available in the near future. However, when presentation and application layers are included the working sets will increase with the functionality. “Value-added” layers implementing services such as encryption may become more common and drive working set sizes up. Control and signalling protocols are often built up from several standard layers, with the sum of the parts including more functionality than is strictly necessary. Thus it seems that there will be protocols whose working sets are larger than the average message, and which do not entirely fit in the cache of available processors. LDLP, and concern for code locality in general, will benefit these protocols.

7 Bibliography

1. V. Jacobson, R. Braden, D. Borman, “TCP Extensions for High performance,” RFC 1323, Internet Engineering Task Force (February 1991).
2. M.K. McKusick, K. Bostic, M.J. Karels, J.S. Quarterman. *The Design and Implementation of the 4.4 BSD UNIX Operating System*. Addison Wesley, 1996.
3. R. Sites, ed. *Alpha Architecture Reference Manual* Digital Press, 1992.

4. V. Jacobson. *Efficient Protocol Implementation*. ACM SIGCOMM '90 Tutorial.
5. D. Clark. *Protocol Performance*. ACM SIGCOMM '94 Tutorial.
6. D. Clark, V. Jacobson, J. Romkey, H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pp 23-29, June 1989.
7. D. Clark, D. Tennenhouse. Architectural Considerations for a new Generation of Protocols. In *Proceedings of SIGCOMM '90*.
8. C. Maeda, B.N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *14th ACM Symposium on Operating Systems Principles*, December 1993.
9. M.B. Abbott, L.L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, Vol. 1, No. 5, October 1993.
10. S. McFarling. Program Optimization for Instruction Caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 183-191, April 1989.
11. J.K Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of Summer 1990 USENIX Conf,* pp 247-256, Anaheim, CA, June 1990.
12. M.A Pagels, P. Druschel, L.L. Peterson. Analysis of cache and TLB effectiveness in processing network I/O. Technical Report 94-08, Department of Computer Science, University of Arizona. March 1994.
13. J.B. Chen, B.N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles,* pp 120-133, December 1993.
14. Digital Equipment Corporation. *Cord*, 1991.
15. A. Eustace, A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Technical Conference* , pp 303-314, Jaunary, 1995.
16. P. Druschel. *Operating System Support for High-Speed Networking*. Dissertation in Department of Computer Science, University of Arizona.
17. D. Mosberger, L.L. Peterson, S. O'Malley. *Protocol Latency: MIPS and Reality.* TR 95-02 from University of Arizona, Department of Computer Science.
18. N.C Hutchinson and L.L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*. 17(1) 64-75, Jan 1991.
19. M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. December 1995.
20. V. Paxson, S. Floyd. Wide-area traffic: The failure of Poisson modeling. In *Proceedings of SIGCOMM '94,* pp 257-268, 1994.
21. W.E. Leland, M.S. Taqqu, W. Willinger, D.V. Wilson. On the self-similar nature of Ethernet traffic. In *Proceedings of SIGCOMM '93,* Vol 23, pp 183-193, 1993.
22. M.S. Lam, L.E. Rothberg, M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Fourth International Conference of Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. Palo Alto, CA, Apr 1991.
23. G.H. Golub, C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989
24. T.A. Dutton et al. The Design of the DEC 3000 AXP Systems, Two High-Performance Workstations. *Digital Technical Journal,* Vol 4(4) 1992.