Cache Behavior of Network Protocols

Erich Nahum, David Yates, Jim Kurose, and Don Towsley*
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{nahum,yates,kurose,towsley}@cs.umass.edu

Abstract

In this paper we present a performance study of memory reference behavior in network protocol processing, using an Internet-based protocol stack implemented in the *x*-kernel running in user space on a MIPS R4400-based Silicon Graphics machine. We use the protocols to drive a validated execution-driven architectural simulator of our machine. We characterize the behavior of network protocol processing, deriving statistics such as cache miss rates and percentage of time spent waiting for memory. We also determine how sensitive protocol processing is to the architectural environment, varying factors such as cache size and associativity, and predict performance on future machines.

We show that network protocol cache behavior varies widely, with miss rates ranging from 0 to 28 percent, depending on the scenario. We find instruction cache behavior has the greatest effect on protocol latency under most cases, and that cold cache behavior is very different from warm cache behavior. We demonstrate the upper bounds on performance that can be expected by improving memory behavior, and the impact of features such as associativity and larger cache sizes. In particular, we find that TCP is more sensitive to cache behavior than UDP, gaining larger benefits from improved associativity and bigger caches. We predict that network protocols will scale well with CPU speeds in the future.

1 Introduction

Cache behavior is a central issue in contemporary computer system performance. The large gap between CPU and memory speeds is well-known, and is expected to continue for the forseeable future [17]. *Cache memories* are used to bridge this gap, and multiple levels of cache memories are typical in contemporary systems. Many studies have examined the memory reference behavior of application code, and recently work has appeared studying the cache behavior of operating systems. However, little work has been done to date exploring the impact of memory reference behavior on network protocols. As networks become ubiquitous, it is important to understand the interaction of network protocol software and computer hardware. Thus, rather than examining an application suite such as the SPEC 95 benchmarks, the workload that we study is network protocol software.

We wish to address the following research issues:

- What is the memory reference behavior of network protocol code? What are the cache hit rates? How much time is spent waiting for memory?
- Which has a more significant impact on performance, instruction references or data references?
- How sensitive are network protocols to the cache organization? How do factors such as cache size and associativity affect performance?
- What kind of impact will future architectural trends have on network protocol performance?

We use *execution-driven* simulation to answer these questions, by using an actual network protocol implementation that we run both on a real system and on a simulator. We have constructed a simulator for our MIPS R4400-based Silicon Graphics machines, and taken great effort to *validate* our simulator, i.e., to ensure that it models the performance costs of our platform accurately. We use the simulator to analyze a suite of Internet-based protocol stacks implemented in the *x*-kernel [20], which we ported to user space on our SGI machine. We characterize the behavior of network protocol processing, deriving statistics such as cache miss rates, instruction use, and percentage of time spent waiting for memory. We also determine how sensitive protocol processing is to the architectural environment, varying factors such as cache size and associativity, and we predict performance on future machines.

We show that network protocol software is very sensitive to cache behavior, and quantify this sensitivity in terms of performance under various conditions. We find that protocol memory reference behavior varies widely, and that instruction cache behavior has the greatest effect on protocol latency in most cases. We present the upper bounds on performance improvements that can

* This research supported in part by NSF under grant NCR-9206908, and by ARPA under contract F19628-92-C-0089. Erich Nahum was supported by a Computer Measurement Group Fellowship and is currently with the IBM T.J. Watson Research Center. David Yates was the recipient of a Motorola Codex University Partnership in Research Grant and is currently with the Boston University Computer Science Department.

SIGMETRICS '97, Seattle, Washington, USA

be expected by improving memory behavior, and the impact of features, such as associativity and larger cache sizes, on performance. In particular, we find that TCP is more sensitive to cache behavior than UDP, gaining larger benefits from improved associativity and bigger caches. We predict that network protocol performance will scale with CPU speed over time.

The remainder of this paper is organized as follows: In Section 2 we describe our experimental environment in detail, including protocols and the execution-driven simulator. In Section 3 we present a baseline cache memory analysis of a set of network protocol stacks. In Section 4 we show how sensitive network protocols are to architectural features such as cache size and associativity. In Section 5 we give an example of improving instruction cache behavior. In Section 6 we outline related work. In Section 7 we summarize our conclusions and discuss possible future work.

2 Experimental Infrastructure

In this section we describe our architectural simulator, network protocol workload, experimental methodology, and validation of the simulator for the workload.

2.1 Architectural Simulator

In order to investigate the memory reference behavior of network protocols, we have designed and implemented an architectural simulator for our Silicon Graphics machine. We use this simulator to understand the performance costs of our network protocol stacks, and to guide us in identifying and reducing bottlenecks. The primary goal of the simulator has been to *accurately* model CPU and memory costs for the SGI architecture.

Our architectural simulator is built using MINT [38], a toolkit for implementing multiprocessor memory reference simulators. MINT interprets a compiled binary directly and executes it, albeit much more slowly than if the binary was run on the native machine. This process is called *direct execution*. MINT is designed to simulate MIPS-based multiprocessors, such as our SGI machines, and has support for the multiprocessor features of IRIX. Unlike some other simulators, it does not require all source for the application to be available, and does not require changing the application for use in the simulator. This means the exact same binary is used on both the actual machine and in the simulator.

A simulator built using MINT consists of 2 components: a front end, provided by MINT, which handles the interpretation and execution of the binary, and a back-end, supplied by the user, that maintains the state of the cache and provides the timing properties that are used to emulate a target architecture. The front end is typically called a *trace generator*, and the back end a *trace consumer*. On each memory reference, the front end invokes the back end, passing the appropriate memory address. Based on its internal state, the back end returns a value to the front end telling it whether to continue (for example, on a cache hit) or to stall (on a cache miss). We have designed and implemented a back end for use with MINT to construct a uniprocessor simulator for our 100 MHz R4400-based SGI Challenge. Figure 1 shows the memory organization for this

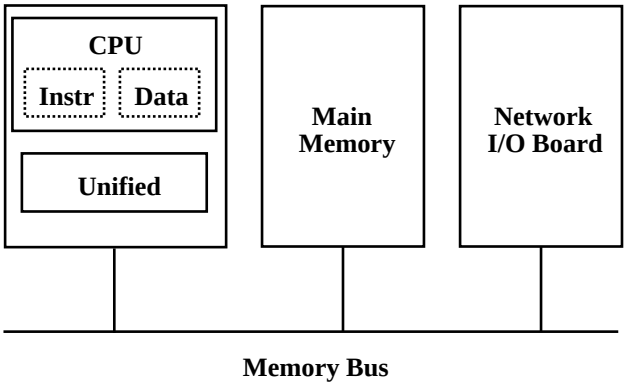

Figure 1: Machine Organization

machine. The R4400 has separate 16 KB direct-mapped on-chip first level instruction and data caches with a line size of 16 bytes. Our SGI machine also has a 1 MB second-level direct-mapped on-board unified cache with a line size of 128 bytes.

The simulator captures the cost of the important performance characteristics of the SGI platform. It supports multiple levels of cache hierarchy, including the inclusion property for multi-level caches [1], and models the aspects of the MIPS R4400 processor that have a statistically significant impact on performance, such as branch delays and load delay pipeline interlocks. It does not, however, capture translation lookaside bufffer (TLB) behavior[1].

As mentioned earlier, we use our simulator to evaluate the cache memory behavior of network protocols. We now present the protocols and test environment that we use. Validation results are given in Section 2.3.

2.2 Network Protocol Workload

The network protocol stacks we consider in this paper are implemented in the *x*-kernel [20], an environment for quickly developing efficient network protocol software. Unfortunately, we did not have access to the source code of the IRIX operating system that runs on our Silicon Graphics machines. Our stack is thus a user-space implementation of the *x*-kernel that we ported to the SGI platform. The code is the uniprocessor base for two different multiprocessor versions of the *x*-kernel [30, 39].

The protocols we examine are from the core TCP/IP suite, those used in typical Internet scenarios. The execution paths we study are those that would be seen along the common case or "fast path" during data transfer of an application. We do not examine connection setup or teardown; in these experiments, connections are already established.

TCP is the Transmission Control Protocol used by Internet applications that require reliable service, such as file transfer, remote login, and HTTP. It provides a connection-oriented service with reliable, in-order data delivery, recovers from loss, error, or duplication,

[1] An earlier version did model the TLB, but we found that the impact on accuracy for this workload was negligible, and that the execution time of the simulator was tripled.

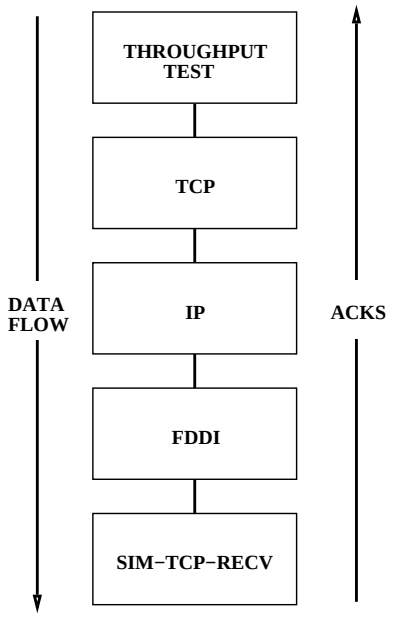

Figure 2: TCP Send-Side Configuration

and has built in flow control and congestion control mechanisms. UDP is a connectionless datagram transport protocol that provides little beyond simple multiplexing and demultiplexing; it does not make guarantees about ordering, error control, or flow control. IP is the network-layer protocol that performs routing of messages over the Internet. FDDI is the Fiber Distributed Data Interface, a 100 Mbit fiber-optic token-ring based LAN protocol.

Our TCP implementation is based upon the *x*-kernel's adaptation of the Berkeley Tahoe release, which we also updated to be compliant with the BSD Net/2 [24] software. In addition to adding header prediction, this involved updating the congestion control and timer mechanisms, as well as reordering code in the send side to test for the most frequent scenarios first [21][2]. In addition, the code has some BSD 4.4 fixes, but none of the RFC1323 extensions [7].

Checksumming has been identified as a potential performance issue in TCP/UDP implementations. Certain network interfaces, such as SGI's FDDI boards, have hardware support for calculating checksums that effectively eliminate the checksum performance overhead. However, not all devices have this hardware support. To capture both scenarios, we run experiments with checksumming on and off, to emulate checksums being calculated in software and hardware, respectively. For our software checksum experiments, the checksum code we use is the fastest available portable algorithm that we are aware of, which is from UCSD [23].

Since our platform runs in user space, accessing the FDDI adaptor involves crossing the IRIX socket layer and the user/kernel boundary, which is prohibitively expensive. Normally, in a user-space implementation of the *x*-kernel, a simulated device driver is configured below the media access control layer (in this case, FDDI). The simulated driver uses the socket interface to emulate a network device, crossing the user-kernel boundary on every packet. Since we wish to measure only our protocol processing software, we replaced the simulated driver with in-memory device drivers for both the TCP and UDP protocol stacks, in order to avoid this socket-crossing cost. The drivers emulate a high-speed FDDI interface, and support the FDDI maximum transmission unit (MTU) of slightly over 4K bytes. This approach is similar to those taken in [4, 16, 34].

In addition to emulating the actual hardware drivers, the in-memory drivers also simulate the behavior of a peer entity that would be at the remote end of a connection. That is, the drivers act as senders or receivers, producing or consuming packets as quickly as possible, to simulate the behavior of simplex data transfer over an error-free network. To minimize execution time and experimental perturbation, the receive-side drivers use pre-constructed packet templates, and do not calculate TCP or UDP checksums. Instead, in experiments that use a simulated sender, checksums are calculated at the transport layer, but the results are ignored, and assumed correct.

Figure 2 shows a sample protocol stack, in this case a send side TCP/IP configuration. In this example, a simulated TCP receiver sits below the FDDI layer. The simulated TCP receiver generates acknowledgment packets for packets sent by the TCP protocol above. The driver acknowledges every other packet, thus mimicking the behavior of Net/2 TCP when communicating with itself as a peer. Since spawning a thread is expensive in user space in IRIX, the driver "borrows" the stack of a calling thread to send an acknowledgment back up.

The TCP receive-side driver (i.e., simulated TCP sender) produces packets in-order for consumption by the actual TCP receiver, and flow-controls itself appropriately using the acknowledgments and window information returned by the TCP receiver. Both simulated TCP drivers also perform their respective roles in setting up a connection.

Our test environment is meant to measure protocol processing time in the network subsystem on the host; it does not measure external factors such as latency across the wire to a remote host, or the effects of network congestion. One of our main performance metrics is *latency*. In our experiments, we define latency as the *total processing time* required for the network protocol code. Latency is the total time between when a packet is sent at the top of the protocol stack and when the send function returns. It thus includes procedure call return time from after a packet is delivered to a device. In our experiments, the reported times for latencies are the average of ten runs, where each run in turn measures the average latency observed over a 5 second sampling interval after a 5 second warmup. During these intervals, other processing can occasionally occur, such as the *x*-kernel's periodic event manager which runs every 50 milliseconds, or the TCP 200 millisecond fast timer. However, we have observed the times for these other events to be statistically insignificant in our experiments.

2.3 Validating the Simulator

In order to validate the performance accuracy of the simulator, a number of benchmarks were run on both the real and simulated ma-

[2] We use 32 bits for the flow-control windows; see [30] for more details.

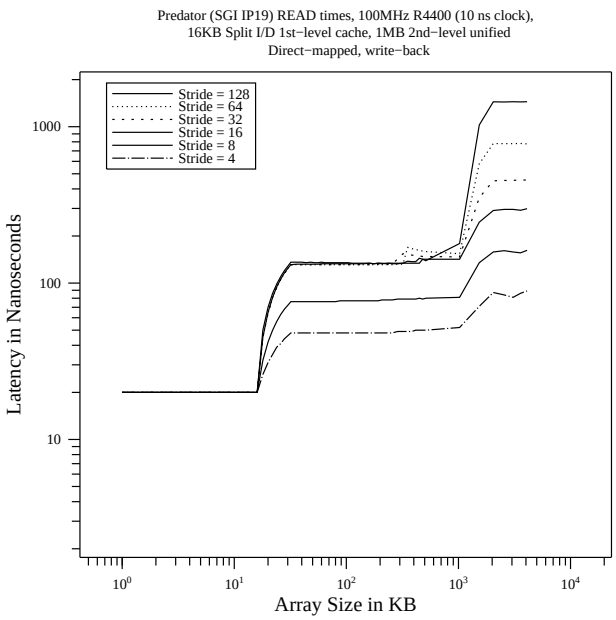

Figure 3: Actual Read Latencies

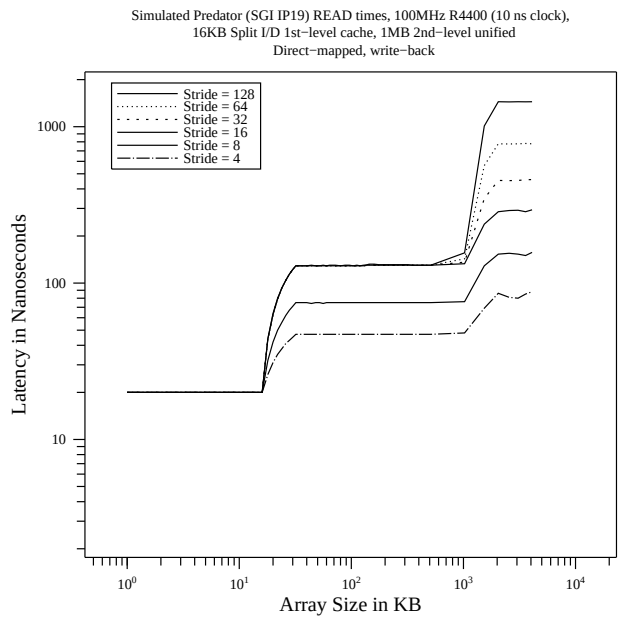

Figure 4: Simulated Read Latencies

Layer in Hierarchy	Read time	Write time
L1 Cache	0	0
L2 Cache	11	11
Challenge Bus	141	147

Table 1: Read and write times in cycles

Benchmark	Simulated	Real	Error (%)
TCP Send Cksum Off	76.63	78.58	-2.48
TCP Send Cksum On	147.84	146.66	0.81
UDP Send Cksum Off	18.43	15.97	15.40
UDP Send Cksum On	71.99	70.30	2.41
TCP Recv Cksum Off	58.06	62.65	-7.33
TCP Recv Cksum On	190.47	198.39	-3.99
UDP Recv Cksum Off	33.80	32.84	2.95
UDP Recv Cksum On	161.78	158.84	1.85
Average Error			4.65

Table 2: Macro benchmark times (μsec) and relative error

chines. We used the memory striding benchmarks from LMBench [25] to measure the cache hit and miss latencies for all three levels of the memory hierarchy: L1, L2, and main memory. Table 1 lists the cycle times to read and write the caches on the 100MHz SGI Challenge. Figure 3 shows the LMBench read memory access time as a function of the area walked by the stride benchmark, as run on our 100 MHz R4400 SGI Challenge. We call this graph a *memory signature*. The memory signature illustrates the access times of the first level cache, the second-level cache, and main memory. When the area walked by the benchmark fits within the first level cache (i.e., is 16 KB or less), reading a byte in the area results in a first-level cache hit and takes 20 nanoseconds. When the area fits within the second level cache (i.e., is between 16 KB and 1 MB in size), reading a byte results in a second-level cache hit and takes 134 nanoseconds. If the area is larger than 1 MB, main memory is accessed, and the time to read a byte is 1440 nanoseconds. Note that the scales in Figure 3 are logarithmic on both the x and y axes.

These memory latency measurements in turn gave us values with which to parameterize the architectural simulator. The same memory stride programs were then run in the simulator, to ensure that the simulated numbers agreed with those from the real system. Figure 4 shows the memory signature of the same binary being run on the simulator for the same machine. As can be seen, the simulator models the cache memory behavior very closely.

While reassuring, these memory micro-benchmarks do not stress other overheads such as instruction costs. What we are most interested in is how accurate our simulator is on our workload, namely, network protocol processing. Table 2 presents a set of protocol processing benchmarks, with their corresponding real and simulated latencies in microseconds, and the relative error. Error is defined as

$$Error = \frac{(Simulated\ Value \quad Real\ Value)}{Real\ Value} * 100$$

A negative error means the simulator *underestimates* the real time; a positive value means it *overestimates* the real time. The *average* error is calculated as the mean of the absolute values of the individual errors. This is to prevent positive and negative individual values from canceling each other out. Note the average error is under 5 percent, with the worst case error being about 15 percent. We are

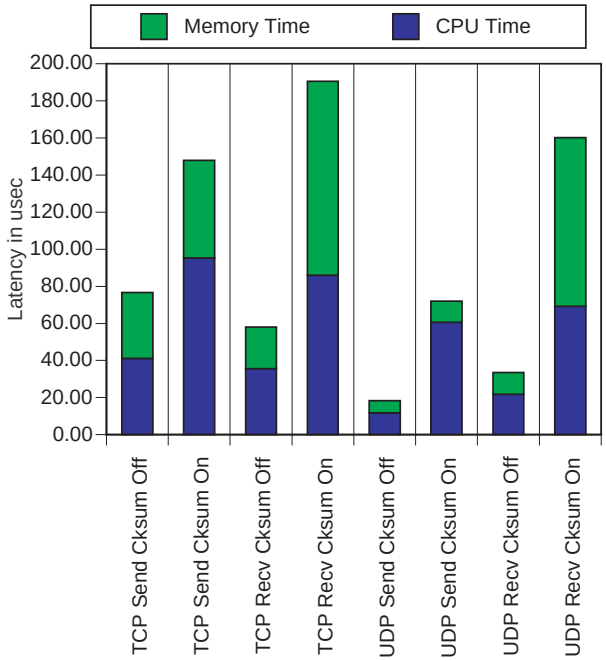

Figure 5: Baseline Protocol Latencies

aware of only a very few pieces of work that use trace-driven or execution-driven simulation that actually validate their simulators [3, 8, 12]. Our accuracy is comparable to theirs. More details about the construction and validation of the simulator can be found in [29].

3 Characterization and Analysis

In this section, we present our characterization and analysis of memory reference behavior of network protocols under a number of different conditions.

3.1 Baseline Memory Analysis

We begin by determining the contribution to packet latency that is due to waiting for memory. Our baseline latencies are produced by executing the core suite of protocols on the architectural simulator, examining TCP and UDP, send and receive side, and with checksumming on and off. Figure 5 shows the latencies in microseconds, distinguishing time spent in computation (CPU time) from time spent waiting for memory (memory time). Table 3 lists the corresponding cache miss rates for the L1 instruction cache, L1 data cache, and L2 unified cache for each configuration.

Studying the data in more detail, we see that all the configurations spend time waiting for memory, ranging from 16 to nearly 57 percent. TCP generally spends a larger fraction of time than UDP waiting for memory, and receivers are slightly more memory bound than senders. UDP generally exhibits low miss rates for all

Protocol Configuration	Level 1 Instr	Level 1 Data	Level 2 Unified
TCP Send Cksum Off	8.30%	5.90%	0.00%
TCP Send Cksum On	3.60%	7.60%	0.00%
TCP Recv Cksum Off	7.00%	2.80%	0.00%
TCP Recv Cksum On	2.80%	15.60%	5.70%
UDP Send Cksum Off	4.00%	0.30%	6.70%
UDP Send Cksum On	1.10%	1.10%	2.50%
UDP Recv Cksum Off	4.70%	1.60%	2.60%
UDP Recv Cksum On	1.50%	17.80%	9.20%

Table 3: Cache Miss Rates for Baseline Protocols

the cache levels, while TCP tends to have higher miss rates for the corresponding experiments, particularly in the data cache on the send side. Experiments that include checksumming generally show lower instruction cache miss rates, since the checksum code is an unrolled loop, and thus exhibits higher temporal and spatial locality. The checksum code also does a good job of hiding memory latency since the unrolling allows checksum computation to overlap with load instructions.

3.2 Hot vs. Cold Caches

The experiments we have presented thus far have involved *hot caches*, where successive packets benefit from the resulting state left by their predecessors. However, the state of the cache can vary depending on application and operating system behavior. For example, when a packet arrives at an idle machine, it is not certain whether the network protocol code or data will be cache-resident. To examine the impact of the cache state on network protocol performance, we ran additional experiments that measure the extremes of cache behavior, using *cold caches* and *idealized caches*.

In experiments with *cold* caches, after processing each packet, the cache in the simulator is flushed [3] of all contents. Each packet is thus processed with no locality benefits from the previous packet. Cold cache experiments thus measure the potential worst-case memory behavior of protocol processing.

In experiments with *idealized* caches, the assumption is made that all references to the level 1 caches *hit*; i.e., no gap between memory and CPU speeds exists. However, the processor model remains the same, as described in Section 2.1. The idealized cache experiments thus give us an unrealizable best-case behavior, and provide upper bounds on performance when focusing solely on the memory behavior of network protocols.

Table 4 presents a sample of results, in this case for the UDP and TCP send sides with and without checksumming. In general, we observe a factor of 5 to 6 increase in latency between experiments with hot caches and those with cold caches. Our experiments using UDP exhibit an increase by a factor of 6, which is even more drastic than the increase measured by Salehi *et al.* [33], who observed a slowdown by a factor of 4 when coercing cold-cache behavior with UDP without checksumming. In experiments using TCP, which they

[3] This flush takes 0 cycles of simulated time.

Protocol Configuration	Cold	Hot	Ideal
TCP Send Cksum Off	375	77	42
TCP Send Cksum On	517	147	96
UDP Send Cksum Off	123	18	12
UDP Send Cksum On	262	71	12

Table 4: Latencies with Cold, Hot, and Idealized Caches (μsec)

Protocol Configuration	Level 1 Instr	Level 1 Data	Level 2 Unified
TCP Send Cksum Off	20.90%	18.50%	21.30%
TCP Send Cksum On	8.30%	21.70%	17.90%
UDP Send Cksum Off	23.10%	19.90%	28.60%
UDP Send Cksum On	4.30%	23.20%	19.50%

Table 5: Cold Cache Miss Rates

did not examine, we see that latencies increase by a smaller factor of 5. This is because TCP exhibits relatively better instruction cache miss rates than UDP in the cold cache scenario. We believe TCP exhibits better spatial locality because TCP does more instruction processing per packet than UDP. For example, the TCP code makes more frequent use of the message buffer manipulation routines in the *x*-kernel *per-packet* than UDP does. This suggests that TCP's re-use of the message buffer code along the fast path contributes to its relatively better i-cache behavior.

We also saw that cold cache experiments using checksumming did not suffer as much relative slowdown compared with the hot cache equivalents as those experiments that did not use checksumming. This was due to better instruction cache behavior, again since the checksum code exhibits both high temporal and spatial locality.

Table 5 presents the cache miss rates for a sample of cold cache experiments. In general, the miss rate goes up by a factor of 2-5. It is interesting to note that despite the initial cold state of the caches, miss rates are still under 25 percent.

3.3 Instructions vs. Data

Much of the literature on network protocol performance has focused on reducing the number of copies, since touching the data is expensive [2, 11, 13, 15]. However, this work has not made explicit how much of this cost is due to *data references* as opposed to *instruction references*. For example, a copy routine on a RISC platform incurs at least one instruction reference for every data reference, namely, the instruction that loads or stores a piece of data. We therefore wish to understand which cost has a larger impact on performance: instruction references or data references.

To test which type of references is more significant, we implemented two additional simulators. The first was an *ideal d-cache* simulator, where data references always hit in the L1 data cache, but instructions are fetched normally from the instruction cache; thus, there are no data cache misses. The second was a complementary *ideal i-cache* simulator, where there are no instruction cache misses,

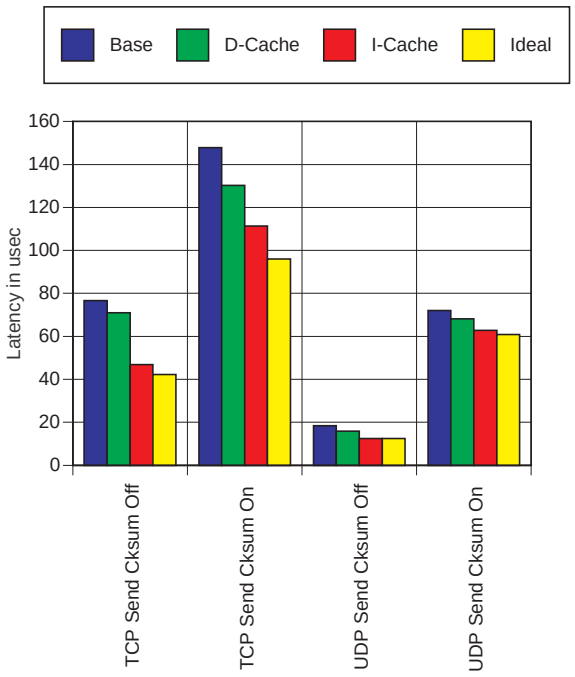

Figure 6: Send Side Latencies

but data references are fetched normally from the data cache. While neither of these simulators represents an attainable machine, they do provide a way of distinguishing instruction costs from data costs, and yield insight into where network protocol developers should focus their efforts for the greatest benefit.

In all of our experiments, the raw number of instruction references exceeds that of data references. In general, tests without checksumming exhibit a roughly 2:1 instruction:data ratio, and experiments with checksumming had a 3:1 ratio. This is consistent between TCP and UDP, and between the send side and receive side. In most of the experiments, instruction references also outweigh data references in terms of their impact on latency. The exception is for protocol architectures that copy data where packets are large. In these experiments, the d-cache was more significant than the i-cache. Figure 6 presents an example of the results, for the TCP and UDP send sides. The columns marked 'D-Cache' are the times using the idealized data cache, and the columns marked 'I-Cache' contain results using the idealized instruction cache. Our results indicate that the performance impact on network protocols of the CPU-memory disparity is felt more through instruction references than through data references. This means protocol developers' efforts are better focused on improving instruction cache behavior than data cache behavior.

4 Architectural Sensitivity

In this section we explore architectural variations in several dimensions, in order to determine how sensitive our protocol performance is to the memory system configuration, and to determine how proto-

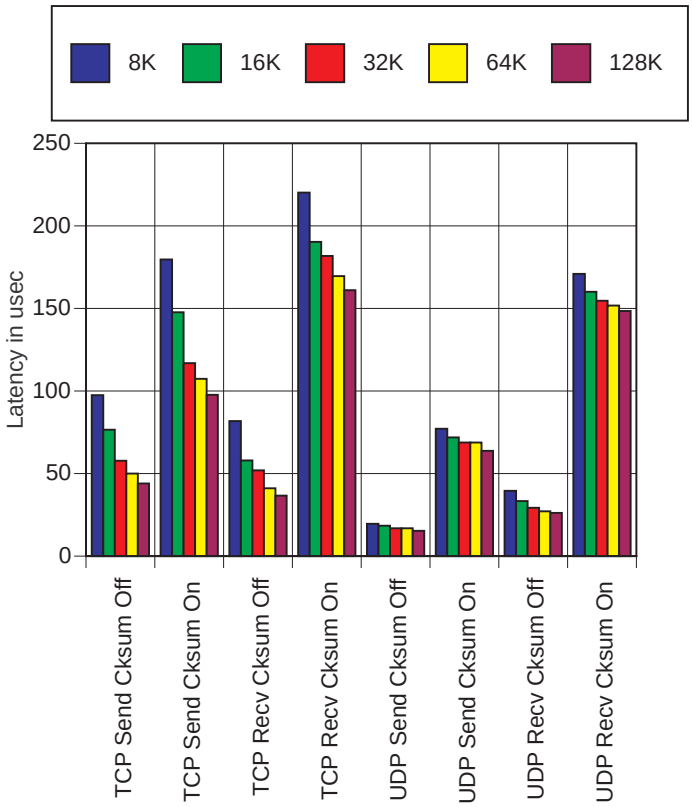

Figure 7: Latencies with Increasing Cache Size

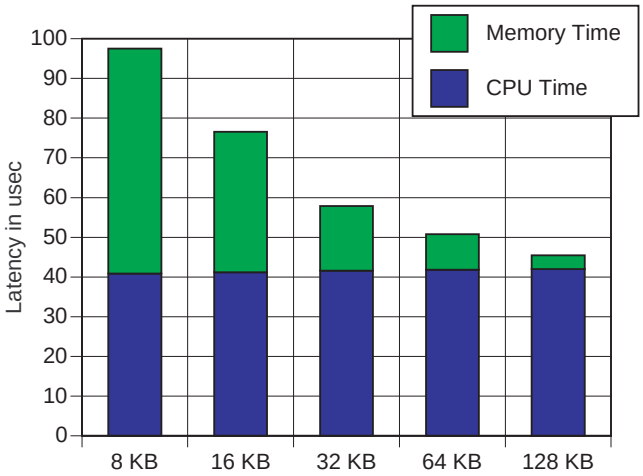

Figure 8: TCP Send Side Latency with Larger Caches

Level 1 Cache Size	Level 1 Instr	Level 1 Data	Level 2 Unified
8 KB	14.40%	7.10%	0.00%
16 KB	8.30%	5.90%	0.00%
32 KB	4.10%	1.90%	0.00%
64 KB	1.90%	1.50%	0.00%
128 KB	0.20%	0.80%	0.00%

Table 6: Miss Rates vs. Cache Size (TCP Send, Cksum Off)

col performance might be expected to change with the introduction of new architectures.

4.1 Increased Cache Size

One trend in emerging processors is increasing transistor counts, which has led to increasing on-chip cache sizes. For example, the MIPS R10000 has 32 KB on-chip caches. As we described earlier, our SGI platform has 16 KB first level caches. While in certain cases (typically the UDP experiments), this cache size produces reasonable miss rates, it is useful to see how sensitive network protocol performance is to the size of the cache. Larger caches, for example, may allow the entire working set of the protocol stack to fit completely in the cache. To evaluate how sensitive network protocol performance is to cache size, we ran a set of experiments varying the first level cache sizes from 8 KB up to 128 KB in powers of two. The level 2 unified cache was left at 1 MB.

Figure 7 presents the latencies for our protocol configurations as a function of the first level cache size. We can observe that increased cache size results in reduced latency, and that TCP is more sensitive to the cache size than UDP. The largest gains comes from increasing the level 1 cache sizes up to 32 KB, with diminishing improvements after that. Figure 8 presents an example in detail, showing TCP send-side latency with checksumming off as a function of cache size, again distinguishing between CPU time and memory time. We can see that the reduction in latency is due to less time spent waiting for memory. Table 6 presents the corresponding cache miss rates. We observe that both the instruction and data cache miss rates improve as the size increases to 128 KB, but that the change in the instruction cache miss rate is more dramatic.

4.2 Increased Cache Associativity

As mentioned earlier, the block placement algorithm for the caches in our SGI machine is direct-mapped, both for the first and second levels. This means that an item loaded into the cache can only be placed into a single location, usually based on a subset of its virtual address bits. If two “hot” items happen to map to the same location, the cache will thrash pathologically. In contrast, TLBs and virtual memory systems are usually fully associative. Cache memories have historically been direct mapped because adding associativity has tended to increase the critical path length and thus increase cycle time [18]. While many RISC machines today have direct-mapped on-chip caches, emerging machines, such as the MIPS R10000, are starting to have 2 way set-associative on-chip caches.

It is thus useful to assess the impact of improved associativity on network protocol performance. To test this, we ran several experiments varying the associativity from 1 to 8 in powers of 2.

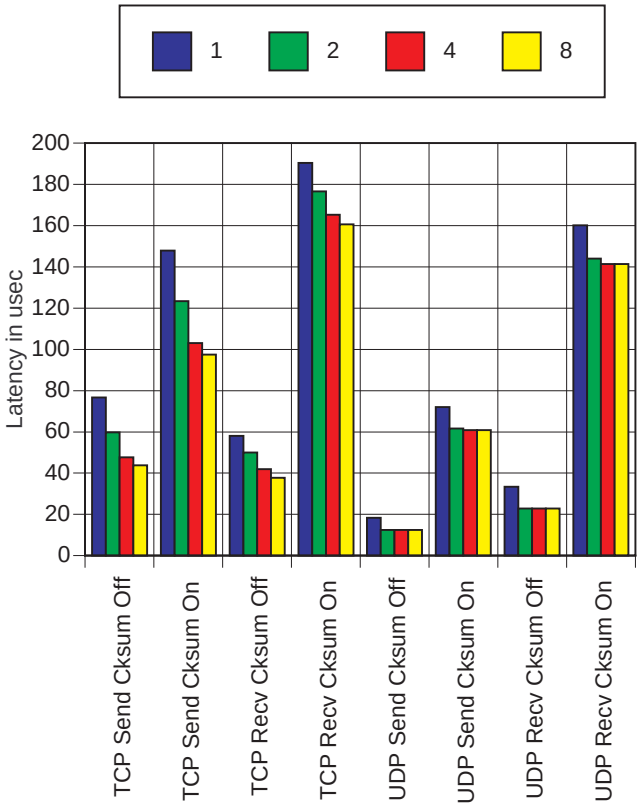

Figure 9: Protocol Latencies with Associativity

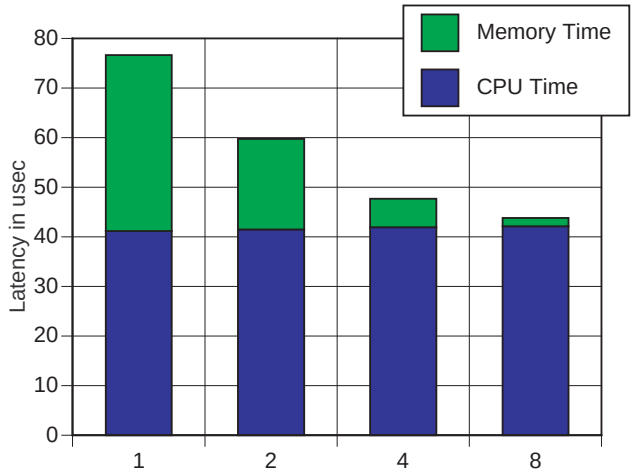

Figure 10: TCP Send Side Latency with Associativity

Cache Assoc.	Level 1 Instr	Level 1 Data	Level 2 Unified
1	8.30%	5.90%	0.00%
2	5.30%	0.40%	0.00%
4	1.70%	0.00%	0.00%
8	0.50%	0.00%	0.00%

Table 7: TCP Send Miss Rates vs. Assoc. (Cksum Off)

While 8-way set-associative on-chip caches are unusual[4], including them in our evaluation helps illustrate how memory time is being spent. For example, it allows us to estimate how much of memory time is due to conflicts in the cache rather than capacity problems [19].

Figure 9 presents the protocol latencies as associativity is varied. In these experiments, for simplicity, all caches in the system have the same associativity, e.g., an experiment marked with 2 indicates that the instruction cache, the data cache, and the level 2 unified cache all have 2-way set associativity. All other factors are held constant, i.e., the first-level cache size remains at 16 KB, and the second-level size remains at 1 MB. We can see that TCP exhibits better latency as associativity is increased all the way up to 8. Figure 10 presents an example in detail, showing TCP send-side latency with checksumming off as a function of set associativity, again distinguishing between CPU time and memory time. We can see that the reduction in latency is due to a decrease in the time spent waiting for memory. Table 7 presents the corresponding cache miss rates. We see that the data cache achieves close to zero misses with 2 way set-associativity, but that the instruction cache miss rates improve all the way up to 8-way. This implies that the Berkeley-derived TCP code has conflicts on the fast path, and that restructuring the code for better cache behavior promises performance improvements. We present one example of this restructuring in Section 5.

In contrast, we do not observe any performance gains for UDP beyond 2 way set-associativity. This is because with 2-way set-associative caches, the UDP stacks achieve very close to a zero miss rate, i.e., they can fit completely within the cache. This shows that UDP has fewer conflicts than TCP, and implies that the opportunity for improving UDP cache behavior is smaller than that for TCP.

4.3 Future Architectures

Given that CPU's are roughly doubling in performance every 2 years, we would like to gain an understanding of how future architectural trends will impact network protocol performance. We have seen that both increased associativity and larger cache sizes improve latency. However, these previous experiments have held the clock speed and miss penalties constant, which ignores two significant trends in computer architecture. First, processor clock speeds are increasing, and second, the gap between memory speeds and CPU speeds is growing.

To gain a better understanding of how network protocol workloads might behave on future architectures, we compared the performance of our stacks on 3 different virtual machines, representing characteristics of 1994, 1996, and 1998, respectively. The 1994 machine is our baseline case, described earlier. The 1996 machine has a faster 200 MHz clock and larger on-chip caches with 2 way

[4] The PowerPC 620 has 8-way on-chip set-associative caches.

Machine (year)	1994	1996	1998
Clock Speed (MHz)	100	200	400
L1 Cache Size (KB)	16	32	64
L1 Associativity	1	2	2
L1 Read (cycles)	0	0	0
L1 Write (cycles)	0	0	0
L2 Cache Size (KB)	1024	1024	1024
L2 Associativity	1	2	2
L2 Read (cycles)	11	13	16
L2 Write (cycles)	11	13	16
Memory Read (cycles)	141	201	300
Memory Write (cycles)	147	275	400

Table 8: Machine Characteristics

set-associativity. It also has larger miss penalties, the values of which we take from the newer SGI IP22 Indigo/2 workstations with a 200 MHz clock. The 1998 machine is an extrapolation of the cycle time and latency trends from the 1994 to the 1996 machine. Table 8 lists the relevant parameters for the 3 machines.

Table 9 presents the latencies of several protocol configurations being run on the 3 machines. As can be seen, latencies fall as CPU's get faster. However, the more important question is, how does network protocol processing scale with processor speed? For that answer, we must normalize by the clock speed and look at the cycles per instruction, or CPI. CPI is a standard measure of architectural performance; an idealized architecture will have a CPI of one[5]. Table 10 shows the relative CPI's for the same set of experiments. In general, we see that the CPI falls as processors get faster. This is because the workload starts to fit within the caches and run at essentially the processor speed. Cold cache experiments are also listed in Tables 9 and 10. We see that the penalty for a cold cache becomes even worse on future machines.

5 Improving I-Cache Performance with Cord

In this section we examine the flip side of hardware-software interaction: tuning or changing the software to take better advantage of the hardware.

In this paper, we have advocated techniques that improve instruction cache behavior. Mosberger *et al.* [27] and Blackwell [5] provide two examples of how this can be done. Mosberger *et al.* examine several compiler-related approaches to improving protocol latency. Using a combination of their techniques (outlining, cloning, and path-inlining), they show up to a 40 percent reduction in protocol processing times. Blackwell [5] also identifies instruction cache behavior as an important performance factor using traces of NetBSD. He proposes a technique for improving processing times for small messages, by processing batches of packets at each layer so as to maximize instruction cache behavior, and evaluates this technique via a simulation model of protocol processing. In this Section we evaluate another technique: improving instruction cache behavior using CORD [35].

[5] Assuming a single-issue processor.

Protocol Configuration	1994	1996	1998
HOT TCP Send Cksum Off	76.61	23.92	8.84
HOT TCP Send Cksum On	147.80	52.44	19.58
HOT UDP Send Cksum Off	18.43	6.27	2.51
HOT UDP Send Cksum On	71.97	30.45	12.18
COLD TCP Send Cksum Off	375.36	247.13	139.20
COLD TCP Send Cksum On	517.59	330.43	181.64
COLD UDP Send Cksum Off	123.81	83.35	47.39
COLD UDP Send Cksum On	262.43	168.08	91.30

Table 9: Machine Latencies (μsec)

Protocol Configuration	1994	1996	1998
HOT TCP Send Cksum Off	2.57	1.58	1.45
HOT TCP Send Cksum On	1.83	1.29	1.21
HOT UDP Send Cksum Off	2.24	1.42	1.42
HOT UDP Send Cksum On	1.30	1.10	1.10
COLD TCP Send Cksum Off	12.87	16.97	23.93
COLD TCP Send Cksum On	6.49	8.29	11.41
COLD UDP Send Cksum Off	16.72	22.62	32.28
COLD UDP Send Cksum On	4.86	6.23	8.48

Table 10: Machine CPIs

CORD is a binary re-writing tool that uses profile-guided code positioning [31] to reorganize executables for better instruction cache behavior. An original executable is run through Pixie [36] to determine its run time behavior and profile which procedures are used most frequently. CORD uses this information to re-link the executable so that procedures used most frequently are grouped together. This heuristic approach attempts to minimize the likelihood that "hot" procedures will conflict in the instruction cache.

We ran our suite of network protocol benchmarks through Pixie and CORD to produce CORDed equivalent executables. Table 11 presents the latencies of both the original and CORDed versions of the programs. As can be seen, the performance improvements range from 0 to 40 percent.

Table 12 presents the cache miss rates for the CORDed benchmarks. Comparing these numbers with Table 3, we can see that the CORDed executables exhibit instruction cache miss rates that are 20-100 percent lower than those for the original executables. In the case of the UDP send side experiment without checksumming, we see that the rearranged executable achieves 100 percent hit rates in both the instruction and data caches! This shows how data references can be indirectly improved by changing instruction references. In this case, the changes have removed a conflict in the L2 unified cache between instructions and data, and subsequently eliminating any invalidations to the L1 caches forced by the inclusion property.

Protocol Configuration	Original time	CORD	Diff (%)
TCP Send Cksum Off	76.61	72.61	5
TCP Send Cksum On	147.80	148.38	0
TCP Recv Cksum Off	58.04	54.33	6
TCP Recv Cksum On	190.42	186.61	2
UDP Send Cksum Off	20.92	12.53	40
UDP Send Cksum On	77.45	65.59	15
UDP Recv Cksum Off	33.46	27.03	19
UDP Recv Cksum On	160.15	148.76	7

Table 11: Baseline & CORDed Protocol Latencies (μsec.)

Protocol Configuration	Level 1 Instr	Level 1 Data	Level 2 Unified
TCP Send Cksum Off	6.10%	6.00%	0.70%
TCP Send Cksum On	3.10%	7.60%	0.80%
TCP Recv Cksum Off	4.70%	2.50%	1.40%
TCP Recv Cksum On	1.70%	15.70%	7.30%
UDP Send Cksum Off	0.00%	0.00%	22.20%
UDP Send Cksum On	0.10%	1.20%	7.70%
UDP Recv Cksum Off	0.60%	1.60%	9.20%
UDP Recv Cksum On	0.20%	17.80%	10.80%

Table 12: Cache Miss Rates for CORDed Protocols

6 Related Work

A number of researchers have addressed related issues in network protocol performance, involving architecture and memory systems. In this section we outline their results and, as appropriate, relate their findings to ours.

Blackwell [5] also identifies instruction cache behavior as an important performance factor using traces of NetBSD on an Alpha. He proposes a technique for improving processing times for small messages, by processing batches of packets at each layer so as to maximize instruction cache behavior, and evaluates this technique via a simulation model of protocol processing.

Clark *et al.* [10] provide an analysis of TCP processing overheads on an Intel i386 architecture circa 1988. Their analysis focuses on protocol-related processing, and does not address OS issues such as buffering and copying data. Their argument is that TCP can support high bandwidths if implemented efficiently, and that major sources of overhead are in data-touching operations such as copying and checksumming. They also note that instruction use of the protocols was essentially unchanged when moving to an unspecified RISC architecture, and that this set is essentially a RISC set. They also focus on data memory references, assuming that instructions are in the cache. We have also focused on protocol-related issues, but on a contemporary RISC architecture, and have quantified the instruction usage. We have examined both instruction and data references, measured cache miss rates for both, and have explored the range of cache behavior.

Jacobson [22] presents a high-performance TCP implementation that tries to minimize data memory references. He shows that by combining the packet checksum with the data copy, the checksum incurs little additional overhead since it is hidden in the memory latency of the copy. We have measured the cache miss rates of protocol stacks of a zero-copy protocol stack on a contemporary RISC-based machine with and without the checksum.

Mosberger *et al.* [27] examine several compiler-related approaches to improving protocol latency. They present an updated study of protocol processing on a DEC Alpha, including a detailed analysis of instruction cache effectiveness. Using a combination of their techniques (outlining, cloning, and path-inlining), they show up to a 40 percent reduction in protocol processing times.

Rosenblum *et al.* [32] present an execution-driven simulator that executes both application and operating system code. They evaluate scientific, engineering, and software development workloads on their simulator. They conclude that emerging architectural features such as lockup-free caches, speculative execution, and out-of-order execution will maintain the current imbalance of CPU and memory speeds on uniprocessors. However, these techniques will not have the same effect on shared-memory multiprocessors, and they claim that CPU memory disparities will become even worse on future multiprocessors. Our workload, in contrast, is network protocol processing, and we have only examined uniprocessor behavior. Although we cannot evaluate some of the more advanced architectural features that they do, our conclusions about our workload on future architectures agree with theirs, due to the increased cache sizes and associativities that are predicted for these machines.

Salehi *et al.* [33] examine scheduling for parallelized network protocol processing via a simulation model parameterized by measurements of a UDP/IP protocol stack on a shared-memory multiprocessor. They find that scheduling for cache affinity can reduce protocol processing latency and improve throughput. Rather than using a model of protocol behavior, we use real protocols to drive a validated execution-driven simulator. We examine both TCP and UDP, determine instruction and memory costs, and vary architectural dimensions to determine sensitivity.

Speer *et al.* [37] describe profile-based optimization (PBO), which uses profiles of previous executions of a program to determine how to reorganize code to reduce branch costs and, to a lesser extent, reduce cache misses. PBO reorders basic blocks to improve branch prediction accuracy and reorganizes procedures so that most frequent call chains are laid out contiguously to reduce instruction cache misses. They show that PBO can improve networking performance by up to 35 percent on an HP PA-RISC architecture when sending single-byte packets. Our work, in contrast, separates the benefits of branch prediction from instruction reordering, and shows that the latter has at least as much of an effect as the former.

Much research has been done supporting high-speed network interfaces, both in the kernel and in user space [2, 6, 11, 13, 14, 15, 26]. A common theme throughout this body of work is the desire to reduce the number of data copies as much as possible, as naive network protocol implementations can copy packet data as much as five times. As a consequence, single-copy and even "zero-copy" protocol stacks have been demonstrated [9, 28]. These pieces of work focus on 'reducing work' done during protocol processing,

namely reducing the number of instructions executed. Our protocol stacks emulate zero-copy stacks. Our results not only measure the cache miss rates and determine the architectural sensitivity, but also distinguish between instruction memory references and data memory references.

7 Conclusions and Future Work

In this paper we have examined cache behavior of network protocols. We summarize our findings as follows:

- *Instruction cache behavior is significant.* Despite previous work's emphasis on reducing data references (for example, in ILP), we find that instruction cache behavior has a larger impact on performance in most scenarios than data cache behavior. Given the spread of zero-copy architectures, and the fact that average packets are small, the relative importance of the i-cache behavior should continue to hold.
- *Cold cache performance falls dramatically.* In cases where caches are cold before packet processing, latencies are roughly 6 times longer for UDP and 4 times longer for TCP without checksumming, and 3.5 times longer for each with checksumming.
- *Larger caches and increased associativity improve performance.* We also show that TCP is more sensitive to these factors than UDP. The associativity results demonstrate that many cache misses in network protocols are caused by conflicts in the cache, and that associativity can remove most of these misses.
- *Future architectures reduce the gap.* Network protocols should scale well with clock speed on future machines, except for one important scenario: when protocols execute out of a cold cache.
- *Code layout is effective for network protocols.* Simple compiler-based tools such as CORD that do profile-guided code positioning are effective on network protocol software, improving performance by up to 40 percent, and reducing network protocol software's demands on the memory system.

These results indicate that instruction-cache centric optimizations hold the most promise, even though larger primary caches with small associativities are becoming the norm. They also indicate that efforts to improve i-cache performance of complex protocols such as TCP are worthwhile. However, simpler protocols such as UDP and IP probably do not warrant the effort, in that small amounts of associativity and automated tools such as CORD are sufficient.

For future work, we briefly discuss several possible directions.

There are several important factors in modern computer architecture that we have not yet examined. Multiple instruction issue, non-blocking caches, and speculative execution are all emerging in the latest generations of microprocessors. Evaluating network protocol processing in the presence of these architectural features remains to be done.

Our results have been obtained on a typical RISC microprocessor. Given the widespread commercial adoption of the Intel x86 architecture, a CISC instruction set, it would be interesting to examine cache behavior and instruction set usage on these platforms. We speculate that, given the more compact instruction representation on CISC machines, the data cache would play a more significant role.

Small scale shared-memory multiprocessors are common server platforms. Our simulator could be extended to accurately model multiple processors, and used to evaluate memory system performance of network protocols on shared-memory multiprocessors.

Acknowledgments

Amer Diwan, Kathryn McKinley, Eliot Moss, and Jack Veenstra all contributed to discussions about memory systems and simulation. Special thanks to Jack Veenstra for answering endless questions about MINT. Eliot Moss also provided valuable feedback on earlier drafts of this paper. Dilip Kandlur, Joe Touch, and the anonymous referees contributed useful comments as well.

References

[1] Jean-Loup Baer and Wen-Hann Wang. On the inclusion property for multi-level cache hierarchies. In *Proceedings 15th International Symposium on Computer Architecture*, pages 73–80, Honolulu Hawaii, June 1988.

[2] David Banks and Michael Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[3] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 14–24, Ottawa, Canada, May 1995.

[4] Mats Björkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 74–83, San Francisco, CA, September 1993.

[5] Trevor Blackwell. Speeding up protocols for small messages. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford, CA, August 1996.

[6] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felton, Kai Li, and Malena R. Mesarina. Virtual-memory mapped interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[7] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance. Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992.

[8] Brad Calder, Dirk Grunwald, and Joel Emer. A system level perspective on branch architecture performance. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–206, Ann Arbor, MI, November 1995.

[9] Hsiao-Keng Jerry Chu. Zero copy TCP in Solaris. In *Proceedings of the Winter USENIX Technical Conference*, San Diego, CA, January 1996.

[10] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[11] Chris Dalton, Greg Watson, David Banks, Costas Clamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 11(2):36–43, July 1993.

[12] Amer Diwan, David Tarditi, and Eliot Moss. Memory-system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, 1995.

[13] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols* , London, England, August 1994.

[14] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* , pages 189–202, Asheville, NC, Dec 1993.

[15] Aled Edwards and Steve Muir. Experiences implementing a high-performance TCP in user space. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols* , pages 196–205, Cambridge, MA, August 1995.

[16] Murray W. Goldberg, Gerald W. Neufeld, and Mabo R. Ito. A parallel approach to OSI connection-oriented protocols. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 219–232, May 1993.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1995.

[18] Mark D. Hill. A case for direct mapped caches. *IEEE Computer*, 21(12):24–40, December 1988.

[19] Mark D. Hill and Alan J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[20] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[21] Van Jacobson. Efficient protocol implementation. In *ACM SIGCOMM 1990 Tutorial Notes*, Philadelphia, PA, September 1990.

[22] Van Jacobson. A high performance TCP/IP implementation. In *NRI Gigabit TCP Workshop*, Reston, VA, March 1993.

[23] Jonathan Kay and Joseph Pasquale. Measurement, analysis, and improvement of UDP/IP throughput for the DECStation 5000. In *USENIX Winter 1993 Technical Conference* , pages 249–258, San Diego, CA, 1993.

[24] S. J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[25] Larry McVoy and Carl Staelin. LMBENCH: Portable tools for performance analysis. In *USENIX Technical Conference of UNIX and Advanced Computing Systems*, San Diego, CA, January 1996.

[26] Ron Minnich, Dan Burns, and Frank Hady. The memory-integrated network interface. *IEEE Micro*, 15(1):11–20, February 1995.

[27] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'-Malley. Analysis of techniques to improve protocol processing latency. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford, CA, August 1996.

[28] B.J. Murphy, S. Zeadally, and C.J. Adams. An analysis of process and memory models to support high-speed networking in a UNIX environment. In *Proceedings of the Winter USENIX Technical Conference* , San Diego, CA, January 1996.

[29] Erich M. Nahum. Validating an architectural simulator. Technical Report 96-40, Department of Computer Science, University of Massachusetts at Amherst, September 1996.

[30] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *First USENIX Symposium on Operating Systems Design and Implementation* , pages 125–137, Monterey, CA, November 1994.

[31] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, NY, June 1990.

[32] Mendel Rosenblum, Edouard Bugnion, Stephen A. Herrod, Emmett Witchell, and Anoop Gupta. The impact of computer architecture on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* , Copper Canyon, CO, December 1995.

[33] James D. Salehi, James F. Kurose, and Don Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing. *IEEE/ACM Transactions on Networking*, 4(4):516–530, August 1996.

[34] Douglas C. Schmidt and Tatsuya Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* , Boston, MA, April 1995.

[35] Silicon Graphics Inc. Cord manual page, IRIX 5.3.

[36] Michael D. Smith. Tracing with Pixie. Technical report, Center for Integrated Systems, Stanford University, Stanford, CA, April 1991.

[37] Steven E. Speer, Rajiv Kumar, and Craig Partridge. Improving UNIX kernel performance using profile based optimization. In *Proceedings of the Winter 1994 USENIX Conference* , pages 181–188, San Francisco, CA, January 1994.

[38] Jack E. Veenstra and Robert J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* , Durham, NC, January 1994.

[39] David J. Yates, Erich M. Nahum, James F. Kurose, and Don Towsley. Networking support for large scale multiprocessor servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Philadelphia, Pennsylvania, May 1996.