Cache and TLB Effectiveness in the Processing of Network Data

Michael A. Pagels
Peter Druschel
Larry L. Peterson[1]

TR 93 04

Abstract

This paper considers the question of how effective caches are in processing network I/O. Our analysis shows that operating system structure plays a key role in the caches behavior, with BSD Unix (a monolithic OS) making more effective use of the cache than Mach (a microkernel OS). Moreover, closer inspection shows that several factors contribute to this result, including how TLBs are managed, how scheduling points are interspersed with data accesses, how data is laid out in memory, and how network functionality is distributed between user space and the kernel.

March 9, 1994

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

[1] This work supported in part by National Science Foundation Grants CRC-9102040, NCR-9204390, ARPA Contract DABT63-91-C-0030 and the Hewlett-Packard Company.

1 Introduction

The emergence of high-speed networks may soon increase the network bandwidth available to workstation class machines by two orders of magnitude. Combined with the dramatic increase in CPU performance, these technological advances make possible new classes of applications, such as multimedia workstations and parallel computing on networks of workstations. However, limited memory bandwidth on these workstations stands between the improvements in network bandwidth and the advantage to be gained by application programs [10].

The problem, simply stated, is that network bandwidth is increasing to the point where it is within an order of magnitude of the memory bandwidth available on a desk-top workstation, making it necessary to limit the number of times network data crosses the memory bus if one hopes to deliver network bandwidth through to the application program [12, 4]. With this in mind, much attention has recently been paid to the design of the network interface, and how it interacts with operating system mechanisms designed to reduce the number of times network data is copied [13, 15, 14, 17, 6].

This paper focuses on a related issue: the effectiveness of the cache in processing network data. Block memory operations, common to network data protocol processing, have been shown to be responsible for a large fraction of a system's memory reference cost [2]. Workstations employ caches to bridge the gap between CPU and main memory speeds, and one would hope that they help to reduce the stress network data puts on the memory bus. However, cache design has historically been influenced by application program behavior (e.g., the SPECmark suite); little is understood about the interaction of caches and network I/O. In other words, the question we are addressing is: once network data is brought into the cache, for whatever reason, how much of it is still there when the application is ready to process it?

Note that when we refer to "cache behavior" in this discussion, we are also considering the translation lookaside buffer (TLB), which in many architectures, must contain an appropriate mapping for the cache access to succeed. This is true for all architectures in which the cache is either physically indexed or physically tagged, and includes HP's PA-RISC, Digital's Alpha, and MIPS R3000/4000 architectures.

This paper makes the following contributions. First, it considers the extent to which operating system structure influences cache behavior with respect to network I/O. The main result here is that a monolithic system (BSD Unix) exhibits much better cache behavior than a microkernel-based system (Mach). Second, it examines Mach's behavior more closely, isolating the factors that contribute to Mach's performance and discusses the extent to which these factors can be overcome to produce better cache behavior. Third, it considers BSD Unix's behavior more closely, identifying what BSD Unix does right and evaluating the extent to which these techniques will continue to be relevant as network bandwidth increases.

2 Experimental Method

To quantify the effect the cache has on the network subsystem, we measured the UDP/IP protocol stack running on top of an Ethernet. Although an Ethernet is not considered a high-speed network, data arriving at even 10Mbps is sufficient to expose important behavior in the cache. Also, our measurements concentrate on the receiving side, which is the most difficult to optimize (and understand).

Figure 1, modelled after the BSD implementation, illustrates the basic structure of the UDP/IP protocol

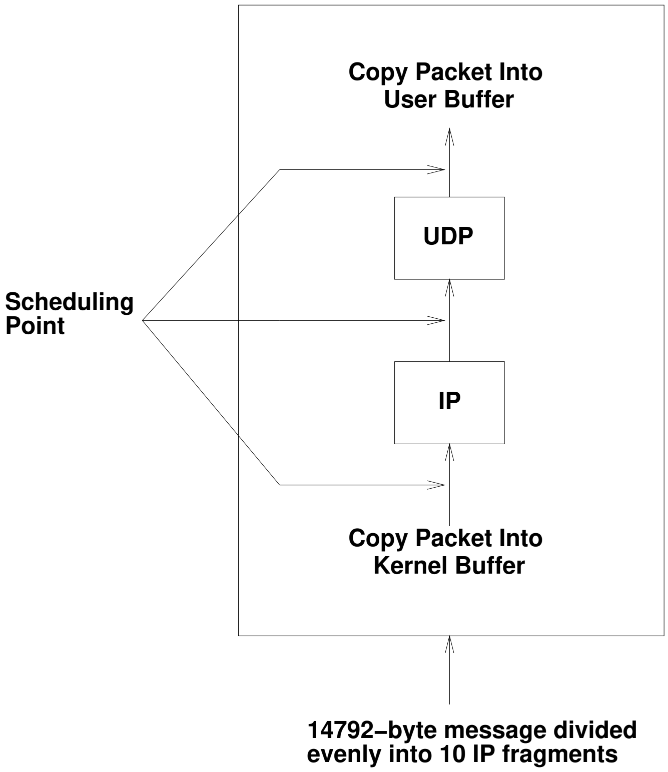

Figure 1: Structure of the UDP/IP Protocol Stack

stack. When an Ethernet packet arrives, the device driver copies it into a kernel buffer (mbuf)[1]. IP then reassembles a set of received packets into an IP datagram, the incoming message, which UDP demultiplexes to the appropriate socket. Eventually, the application program reads the datagram, which causes it to be copied into a user buffer. As denoted in the figure, there are three points at which the CPU is potentially rescheduled: after the interrupt, when IP discovers that more packets (fragments) need to arrive before the datagram is complete, and when the application program is selected to run.

In our study, the sender transmits a sequence of 14792-byte UDP messages to the receiving host. Each of these messages is fragmented into ten equal-sized Ethernet packets, and when they arrive at the receiving host, are stored in ten mbufs. By the time the fragmented message traverses IP and UDP and is placed in the socket queue, it is actually stored as a list of 20 mbufs: ten small mbufs containing mostly headers, and ten large mbufs containing 1460 bytes of data. At the point the user process copies the data from these mbufs into the user buffer—this happens in the *soreceive* routine of the BSD implementation—we measure how long it takes to copy from each of the 1460-byte mbufs that contain data. Assuming the network data were in the cache as a result of the copy into the mbuf, how long the copy out of the mbuf takes tells us about the effectiveness of the cache. This effectiveness is indicative of the benefits an application could obtain in a copy-free implementation; the application would directly obtain this cache residency.

The measurements were taken on a 50MHz HP Apollo 720 workstation [8, 7]. These workstations

[1]For the purpose of this paper, we do not distinguish between 128-byte mbufs, and larger (page-sized) mbuf clusters.

contain a 256KByte data cache and a 128KByte instruction cache. The data cache is write back, direct mapped, virtually addressed and physically tagged with 32-byte cache lines. This design requires an appropriate mapping to exist in the 76 entry data TLB for a successful data cache access. The PA-RISC 1.1 implementation in these workstations provides an interval timer—a user-level accessible register that increments every clock cycle. We simply use this register to count the number of cycles required to copy each mbuf, taking special care to ensure that the data recording machinery does not overly influence the results.

Event	Cycles	Time (ns)
Cache line fill requiring writeback	23	460
Cache line fill not requiring writeback	18	360
Mach TLB entry load	170	3400
BSD TLB entry load	91	1820
HPUX TLB entry load	88	1760

Table 1: Measured OS and Memory Performance

More precisely, we measure how long it takes to *load* the data from the mbuf, not how long it takes to perform the whole copy. In this context, if the data is in the cache, it can be loaded at a rate of one word per cycle. If the data is not in the cache, how long the load takes depends on whether or not the cache line to be filled was dirty—a cache line is dirty if its contents has been altered requiring a write-back to main memory. Table 1 gives the measured cycle count required to fill a cache line for both the dirty and clean case. It also reports the cycle count for loading a TLB entry under the three different operating systems we measured: the University of Utah implementation of Mach with a OSF/1 Unix server (MK69 and server release 1.0.4.B.1), the University of Utah implementation of 4.4 BSD, and HP's commercial Unix offering HPUX 8.07.

We use these three operating systems because their implementation of the UDP/IP protocol stack are all BSD-based, therefore exposing the structural differences of each system. Although similar, there are important differences. The most notable is that in Mach the protocol stack is implemented in the Unix server (a user process, rather than the kernel), the copy at the bottom of the stack is from a Mach message into an mbuf (rather than from the device), and the copy at the top of the stack is from an mbuf into an emulator buffer (rather than into a user buffer).

Finally, we varied the experiment along several additional dimensions. We now summarize these dimensions; the relevance of each is discussed in the following sections.

- **Inter-Message Gap:** We vary the delay between each message transmitted by the sending host.[2] With a sufficiently large inter-message gap, one message is completely processed by the time the next one arrives. With a sufficiently small inter-message gap, the receiving host is saturated. Unless otherwise stated, we use a large gap of 20ms (unsaturated case) and a small gap of 0ms (saturated case). Note that while the saturation point actually falls between 0 and 20ms, these gaps were safely on one side or the other of the saturation point across all other dimensions, and the difference between any two gaps on either side of the saturation point have no effect on the first-order behavior of the system. We discuss the second-order effects when appropriate in subsequent sections.

[2] Note that independent of the inter-message gap, the ten fragments that make up one message are always sent back-to-back.

- **Checksumming Enabled:** UDP has an optional checksum. When enabled, we ensure that the network data is in the cache when UDP completes. Thus, there is only a single scheduling point permitting context switches to effect cache residency between the time the data was known to be in the cache, and when we check to see how much remains in the cache. Unless otherwise stated, checksumming is disabled.
- **TLB Forced:** By touching a word in the same page as an mbuf—but not a word that shares a cache line with the mbuf data—just before measuring how many cycles it takes to copy the data out of the mbuf, we ensure that accessing the mbuf will not cause a TLB miss. Unless otherwise stated, the TLB is not forced.
- **Receive Queue Size:** We adjust the number of UDP messages that can be queued on the socket waiting to be read by the application program. Unless otherwise stated, the queue size is set to one.
- **Load:** We adjust the background job(s) running during the experiment. We have designed an artificial application that walks down each received UDP message, and for each word of the message, accesses a distinct randomly selected state variable. These state variables are blocks of some fixed sized, evenly distributed over a 256 KByte address range, which because of the characteristics of the cache, make them uniformly distributed over the data cache. This applications simulated any program that accesses more data than it reads, compilers and linkers behave in a similar fashion. Unless otherwise stated, no artificial load is placed on the system.

3 Results

This section reports on a series of experiments designed to isolate and quantify the effects of different system features on cache behavior. It begins with a coarse-grained measurement of three different operating systems—Mach, BSD, and HPUX—and then looks at each of the systems in more detail.

3.1 Coarse-Grained Results

We begin by measuring the number of cycles required to read the data about to be copied from the mbufs into the user buffer for each of the three operating systems. In all cases, the default settings described in Section 2 were used, including both 20ms(large) and 0ms(small) inter-message gaps. A timing measurement was taken per mbuf, but only for the large data mbufs. The smaller header mbufs are typically less than a cache line in length; accessing their control information loads their data portions into the cache.

Figures 2 and 3 provide histograms of overhead cycles for the three operating systems with large and small inter-message gaps, respectively. Note that the histograms have different y-axis scaling to improve readability. Overhead cycles are those required beyond the number necessary if the data were in the cache. A zero overhead indicates no cache misses. Less than 1% of the mbufs had overhead times in excess of 2000 cycles. These large overheads can be attributed to context switches or interrupts that occur during a timing interval; these outliers have been excluded from the reported data. In each case, the histogram consists of 3000 mbuf timing measurements. Clearly the histograms portray substantially different cache residency values between the differing operating systems and execution domains.

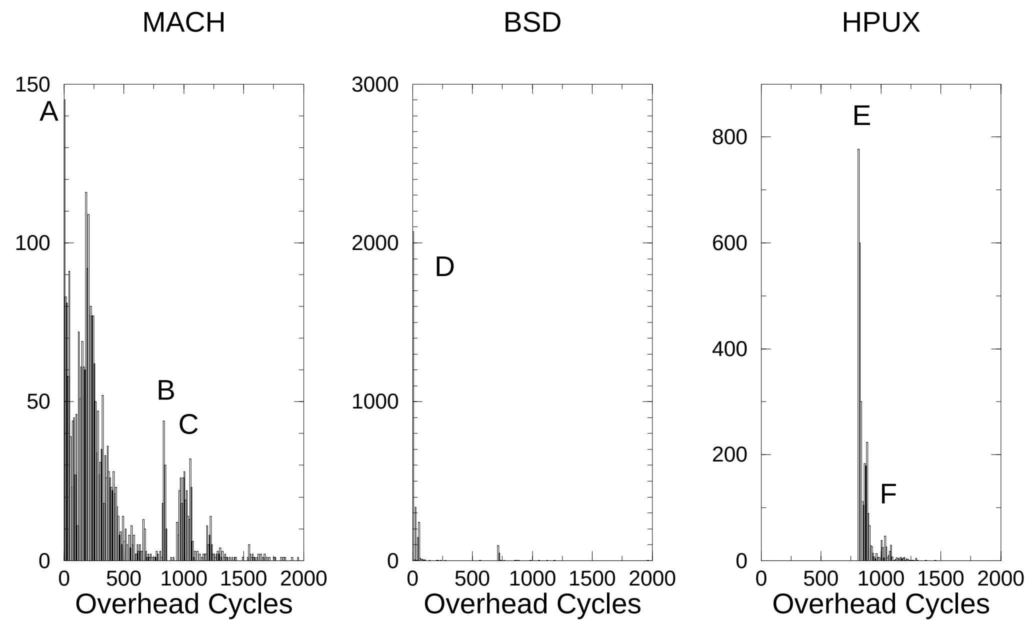

Figure 2: Fragment Overhead Histograms with Large Intermessage Gap

First consider Figure 2, the execution domain where there is a substantial inter-message gap. Mach's peak 'A' indicates that out of 3000 mbufs, 148 of them were found to resided totally in the cache at copy-out time. Accessing that data required zero overhead cycles indicating a 100% cache hit rate. Next consider peak 'B', which occurs at a bin containing values between 840 and 849. Using data from Table 1 and computing that a 1460-byte mbuf contains either 46 or 47 (if the mbuf is not perfectly aligned) cache lines, an mbuf with a 100% miss rate of clean cache lines would require 828 or 846 overhead cycles to access. This suggests that Peak 'B' corresponds to the timing measurements of mbufs that occupied 47 cache lines and experienced a 100% miss rate of clean cache lines. It is important to note that this does not mean that all measurements in this bin were caused by a 100% clean miss rate,[3] just that the measurement is consistent with that conclusion. Performing the same computation with the dirty miss overhead penalty generates overhead values of 1058 and 1081 cycles; peak 'C' corresponds to this value within the Mach histogram. The overhead times computed for 100% clean and dirty miss rates do not change with operating system, they are only architecture dependent.

BSD's histogram is much simpler to describe. The major peak 'D' indicates that 2100 out of 3000 (70%) of the mbufs resided totally within the cache at copy-out time.

Peak 'E' in the HPUX histogram occurs at an overhead time consistent with a 100% clean miss rate on mbuf data. This is because our assumption that all the data was in the cache at the bottom of the protocol stack is wrong—HPUX uses DMA to bring the network data into main memory, but these DMA transfers are not loaded through the cache. The misses in this case are primarily clean due to the light load on the system; there is little activity dirtying the cache as can be seen by the small peak at 'F'.

[3] A combination of a lower clean miss rate plus dirty misses could compute to the same bin.

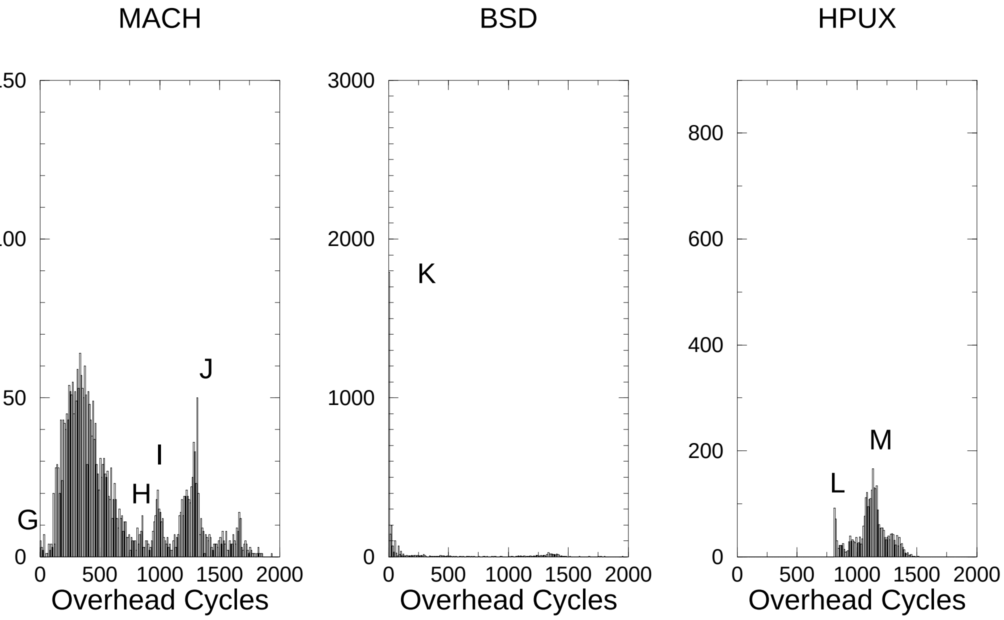

Figure 3: Fragment Overhead Histograms with Small Intermessage Gap

The histograms in Figure 3 represent 3000 mbufs per operating system where the sender transmits messages as fast as possible. Due to the small inter-message gap, the receiver has few cycles with which to process the packet. As the receiver has a single receive buffer in this configuration, there is a substantial dropped packet rate. This execution domain approaches that expected as network bandwidth increases.

Peak 'G' in Mach's graph indicates a substantial decrease in the number of mbufs totally cache resident; 5% in the large gap case to 0.2% in the small gap case. Peaks 'H' and 'I' indicate the 100% clean and dirty miss cases. The most significant change occurs at overheads higher than 100% dirty misses with the peak centered at 'J'. Mbufs with these overheads required more time to access than can be explained by cache fills alone.

BSD continues to exhibit high cache utilization, but there is some decrease in the number of totally cache resident mbufs. Peak 'K' represents a change from 70% of the mbufs totally cache resident with large inter-message spacing, to 60% totally cache resident when the gap in minimized.

The shift in the HPUX histogram indicates that substantial self-interference is occurring within the mbufs. The HP Apollo system is based upon a direct-mapped cache, when data occurring on different virtual pages within the same task map to the same cache line self-interference occurs which replaces the former cache line with the current. If the replaced cache line is still within the working set of the task, a subsequent access may cause repeated replacements. Due to the DMA-based interface device, data is not loaded into the cache early. However, when it is brought into the cache, the peak at 'M' indicates that the required cache lines are typically dirty.

Table 2 summarizes significant data computable from the histograms. This table introduces the metric of "average overhead cycles per word access". An overhead value of 0 indicates that all of the data was found within the cache. A 100% clean miss rate is equivalent to 2.25 overhead cycles per word, and a 100% dirty miss rate implies a 2.875 overhead cycles per word penalty. For example, Mach has a 0.96

Observation	Large Gap			Small Gap		
	Mach	BSD	HPUX	Mach	BSD	HPUX
% mbufs totally within cache	5	70	0	0.2	60	0
% mbufs less than 100% clean miss	83	95	0	71	90	0
% mbufs less than 100% dirty miss	87	99.7	98	73	91	39
% mbufs greater than 100% dirty miss	13	0.3	8	27	9	61
average overhead cycles per word access	0.96	0.13	2.4	1.7	0.45	3.0

Table 2: Summary of Coarse-Grained Results

cycle-per-word penalty in the large inter-message gap case; this is 7 times worse than BSD. HPUX's value of 3.0 is worse than can be expected from cache effects alone—effects other than the cache must have had significant impact. As HPUX's network driver does not initially load network data into the cache, care must be taken when comparing average overhead cycle values with other architectures. HPUX has isolated the majority of its cache effects to copy-out time, thereby providing additional benefits not apparent from this table.

In the following sections, potential non-cache memory system effects contributing to Mach's and HPUX's high overhead numbers are explored, along with with an indepth examination of the details of BSD's network cache behavior.

3.2 Mach

In the previous histograms, both Mach and HPUX contained a number of mbufs whose access times were in excess of that explained by 100% misses on dirty cache lines. There are two additional potential contributors to memory access overheads. First, on the HP Apollo, DMA transfers from other devices compete with the cache for access to memory. If a DMA transaction is underway when a cache miss occurs, the cache line fill will not begin until the DMA transfer has completed. This delay would increase the ideal cache fill times provided in Table 1. Second, the HP Apollo uses a physically tagged cache. In order for a cache line to be accessed, a mapping from the requested virtual address to its physical location must be made prior to cache tag matching. As shown in Table 1, the penalty for a TLB miss can be substantial. Note that Mach's large TLB miss handling time is probably due to the portable nature of portions of the TLB miss handling code; HPUX and BSD have highly tuned machine-specific handlers.

In order to assess the impact of TLB miss handling on cache behavior, data were collected for mbuf access times with the mbuf's virtual to physical mapping forced into the TLB. Care was taken to insure that the TLB forcing operation did not load into the cache any mbuf data to be subsequently timed. Figure 4 provides the histogram of those measurements with both a large and small inter-message gap. The peak at 'N' indicates that substantially more mbufs were found totally within the cache than without TLB forcing—5% without forcing and 11% with forcing. It is important to note that the number of mbufs with times less than that expected with a 100% clean miss rate was essentially equal in both the forced and unforced experiments. This indicates that TLB forcing had no significant impact on the underlying cache miss rate.

The change in height of the peaks occurring at the time expected for 100% dirty cache line misses—peak 'C' in Figure 2 and peak 'P' in Figure 4—and the corresponding increase in the peak at 'O' suggests that most

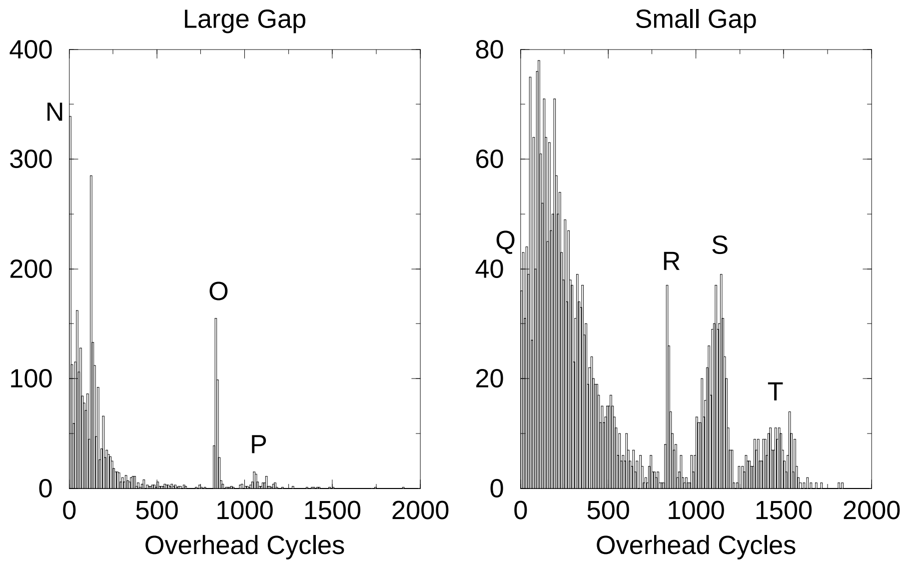

Figure 4: Mach mbufs Forced into TLB

of the mbufs at peak 'C' were probably 100% clean cache line misses plus a TLB miss. This interpretation provides more consistency between the three sets of results.

Peak 'Q', representing the 1% of the mbufs totally contained in the the cache, is nearly a nine fold improvement over peak 'G' in Figure 3. Thus, the high overheads previously seen were not solely the fault of poor cache behavior, TLB miss overhead was a substantial contributor. Moreover the increase in height of peak 'R' indicates that many of the mbufs previously assumed to have experienced 100% dirty misses most likely were 100% clean cache line misses with the addition of a single TLB miss.

A peak at the same location as Figure 3's peak 'J'—corresponding to the expected time of 100% dirty cache line misses plus one TLB miss—does not occur in Figure 4 Instead, these mbuf measurements now occur at peak 'S', corresponding to the expected time of 100% dirty cache line misses plus without a TLB miss. Finally, data lying beyond the time expected with 100% dirty cache misses—peak 'T'—can not be explained by either ideal cache load times or TLB miss handler overheads. We attribute this peak to DMA transfers interfering with the cache for access to the memory system.

With TLB effects accounted for, we now turn our attention to other factors that effect cache efficacy. Specifically, cache residency can be adversely effected primarily in two ways: (1) a task may self-interfere—different addresses are accessed which happen to map to the same cache line, or (2) another task maybe executed whose cache accesses flush the cache lines loaded by the original task [11]. The structure of the UDP/IP stack (see Figure 1) provides a number of scheduling points where another task may adversely effect cache residency at copy-out time.

Due to the scheduling point caused by IP reassembly, mbufs which make up earlier portions of a message can experience more context switches than those later in the message; they are resident in the system awaiting the arrival of the remainder of the message. If the number of context switches which occur between the

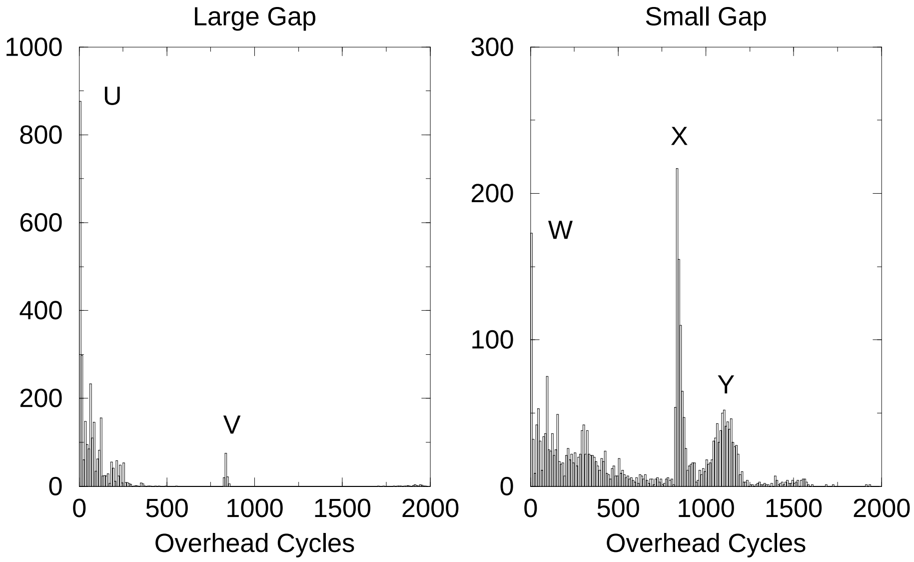

Figure 5: Mach mbufs Forced into TLB with Checksumming

initial copy-in and the copy-out measurement point is significant, the mbufs which make up the earlier portions of the message should have less cache residency than those that make up later portions. Separate distributions were constructed by combining mbuf access timing data based upon the mbufs position within the message—as their were ten mbufs per message, there were ten distributions. The average overhead cycles per word access value for the mbufs that made up the later portions of the message was smaller than for the earlier mbufs. A Kolmogrov-Smirnov test demonstrated that there was no significant difference between the distributions made up of temporally adjacent mbufs, but there was significant ($p < 0.05$) difference between distributions for mbufs at least four apart.

To better isolate the effects of a single context switch we enabled UDP checksumming. Enabling checksumming reduces to one the number of scheduling points that occur between the last cache load of mbuf data and the copy-out point. UDP checksumming entails an access of all data within a UDP message, this access ensures that the data has been placed into the cache. Figure 5 provides histograms of mbuf access time overheads when UDP checksumming is enabled and mbufs are forced into the TLB.

Peaks 'U' and 'W' indicate a substantial increase in those mbufs that are totally within the cache—30% with large inter-message gaps and 6% with small inter-message gaps, versus 11% and 1% without checksumming enabled. With large inter-message gaps, an incoming message can be completely processed prior to the receipt of the next message. Additionally, as the system is lightly loaded, few tasks are available to adversely effect cache residency within a single scheduling point. These factors both contribute to the small number of mbufs experiencing a 100% miss rate. It is still somewhat surprising, as there was but a single intervening context switch point, that not more of the mbufs where found totally within the cache. This can only be attributed to the combination of cache self-interference and a significant light-load cache footprint.

In the small inter-message gap execution domain, there is insufficient time to complete message processing prior to the receipt of the next message. This results in the potential for a number of tasks interleaving at the single scheduling point. The large peak at ‘X’ occurring at the time expected for mbufs with 100% clean cache line misses indicates that other tasks are executing at the scheduling point flushing mbuf data from the cache. Due to the nature of the checksum operation, most of these cache lines are clean. Table 3 summarizes these results.

	No Checksumming		Checksumming	
Observation	Large Gap	Small Gap	Large Gap	Small Gap
% mbufs totally within cache	11	1	30	6
% mbufs less than 100% clean miss	85	72	98	51
% mbufs less than 100% dirty miss	98	81	98	83
% mbufs greater than 100% dirty miss	2	19	2	17
average overhead cycles per word access	0.6	1.3	0.3	1.7

Table 3: Summary of Mach TLB Forced Results

3.3 HPUX

Recall that HPUX’s peak ‘M’ from Figure 3 indicates that a large number of mbufs (61%) experienced access times in excess of that expected from 100% dirty cache line misses. When the mbufs were forced into the TLB using the technique from the previous section, there was no significant change in the histogram. As the number of extraneous cycles is small, the probable cause is delayed cache line loads due to DMA contention for access to main memory.

Figure 6 displays the effects of enabling checksumming on the distribution of mbuf access times. Now that data is forced into the cache due to the checksum algorithm, and the number of scheduling points has been reduced to one, nearly 50% of the mbuf accesses are totally within the cache. Mach, in the same execution domain only had 6% of mbufs totally within the cache; HPUX must have a substantially smaller cache footprint, or substantially less self-interference than Mach.

3.4 BSD

Both TLB forcing and checksum enabled experiments were run using BSD. In the TLB forcing experiment, there was an improvement in the number of mbufs found totally within the cache—from between 70% (large gap) to 60% (small gap) without forcing to 80% (both gaps) with forcing. BSD with checksumming enabled showed a slight decrease in the number of mbufs found totally within the cache when there were large inter-message gaps. Both large and small inter-message gap execution domains had 60% of the mbufs totally within the cache, down from 70% for the large inter-message gap domain. This lack of change in the access time histograms requires that even in the case of heavy network load (small inter-message gaps) and moderate load (checksumming) BSD mbufs experienced little or no self-interference.

The experiments presented up to this point have been overly simplistic in two ways: they assumed a single receive buffer, and the application did not process the incoming data. Although there was no reason

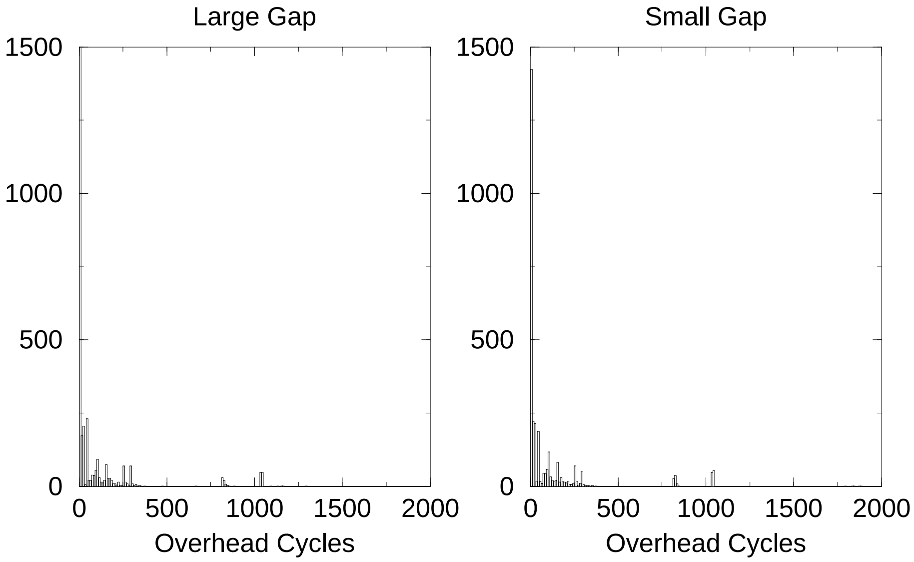

Figure 6: HPUX with Checksumming

to remove these simplifications for Mach and HPUX—both systems put significant stress on the cache even with a single buffer and no load—because BSD performed well under these simplistic conditions, we are interested in learning if it continues to perform well under more realistic conditions. We therefore performed the following two experiments on BSD.

First, to accommodate bursty traffic, most network connections are established with buffering for more than one message. Increased buffering potentially increases that amount of time a message is in the cache prior to copy-out; rather than being dropped the message is buffered. Figure 7 presents the histogram of mbuf access times when the socket buffer has been increased to 246467 bytes, the maximum size allowed in the experimental BSD configuration. This is enough space for 16 experimental messages. The most significant change is the drop in the number of mbufs found totally within the cache when there is a small inter-message gap—from 60% in the previous execution domains to 52% with increased buffer space. This execution domain still reflects a largely unloaded system, and with BSD's limited cache self-interference between mbufs, a small change was expected.

Second, the execution domains considered so far made no use of the incoming message data, this is not typically the case. In this experiment, we simulate a network synchronized load; each word of an incoming message causes the receiving user program to access a randomly located block of n bytes. The starting addresses of these blocks range over 256KBytes, and due to the direct-mapped nature of the HP Apollo cache, will be distributed over the entire cache. This access pattern simulates a program accessing "state" information based upon network data; no other load is applied to the system and the messages are sent with a large inter-message gap.

Figure 8 presents the histogram of mbuf access times with varying state block sizes. With moderate amounts of extra load there is a substantial decrease in BSD cache efficacy. Loads beyond 25 bytes per word

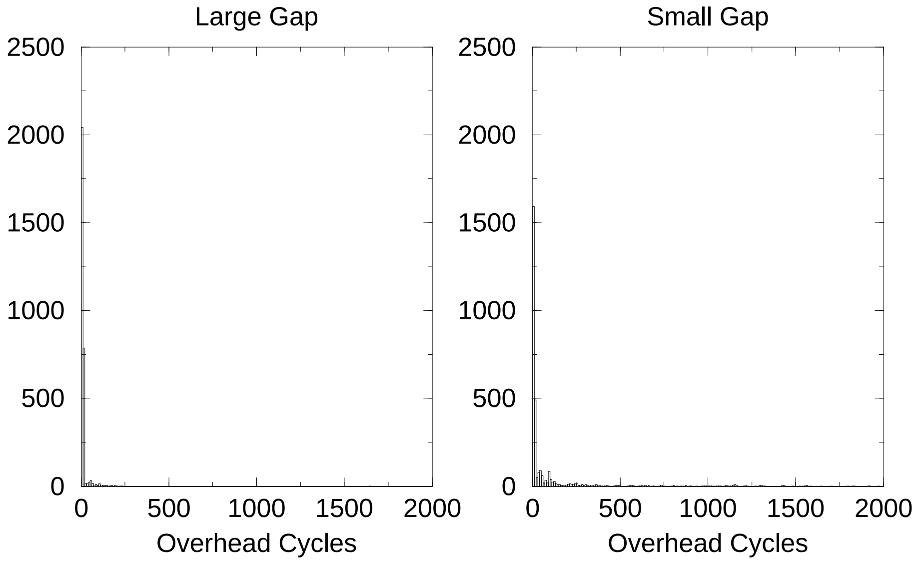

Figure 7: BSD with Large Socket Buffers

induce a non-zero drop rate as the receiver is unable to process incoming packets at a sufficient rate. Table 4 provides a summary. These values are 4.6 to 10 times larger than the unloaded BSD results, and are similar to unloaded Mach results. A 10% increase in the bytes accessed per word induced a 10 fold decrease in BSD's cache utilization; this implies that with increasing network bandwidth BSD can experience a substantial decrease in cache efficacy. The bottom line is that under a heavier load conditions, BSD's cache residency decreases significantly.

Observation	23 bytes / word	24 bytes / word	25 bytes / word
% mbufs totally within cache	5	0.3	0.3
% mbufs less than 100% clean miss	96	92	86
% mbufs less than 100% dirty miss	97	93	90
% mbufs greater than 100% dirty miss	3	7	10
average overhead cycles per word access	0.6	1.1	1.3

Table 4: Summary of BSD Load Results

4 Discussion

TLB misses can have a substantial performance impact on network data. On the HP Apollo the data TLB contains 76 slots covering 304 KBytes of physical memory. Even though this is 1.2 times the size of the cache, network data typically occupies quarter pages, which effectively requires more TLB slots unless careful management reduces scattering. In the case of Mach, TLB misses caused a 50-85% increase

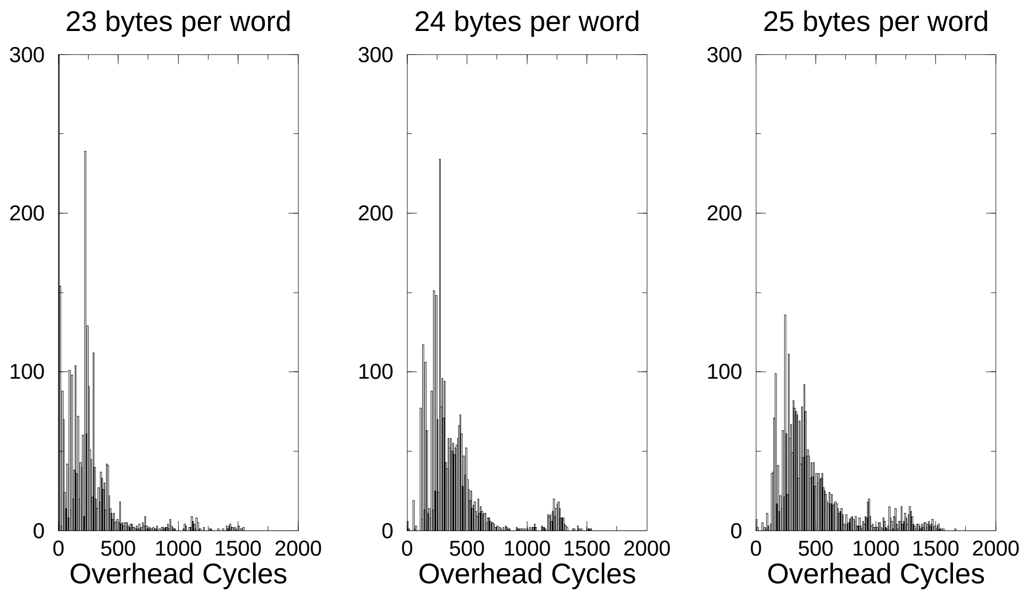

Figure 8: BSD with Increased Load

in the average number of overhead cycles per word access; this is a substantial penalty. Many modern architectures, including the HP Apollo, support "group" or "super" TLB slots. These special TLB slots map many contiguous virtual pages to contiguous physical pages, thereby decreasing the likelihood of a miss as they tend to stay resident longer, and decreasing the number of potential misses for any data within the set of grouped pages. Group TLBs also present special problems as virtual pages are no longer a single size containing a single physical page. None of the operating systems in these experiments used group TLBs to minimize TLB misses.

Checksumming and other forms of system access of network data provides substantial opportunity for self-interference to decrease cache and TLB efficacy. This was not readily apparent from our checksum experiments as all packets were checksummed. Data were not collected where only a portion of the packets required checksum processing. Had this been the case, those packets not being checksummed would have been adversely effected—i.e., flushed from the cache and TLB—by those packets being checksummed. This is an important secondary effect of poor cache utilization—flushing a cache line still in the working set of another process penalizes that process by requiring it to perform a reload on the flushed line. The BSD load experiments provide some insight into this process, however, they are not specifically measuring cache interference. The lack of apparent self-interference within BSD is likely an artifact of our experiments, not necessarily the design of BSD. The BSD kernel allocates mbuf clusters from a large pool potentially scattering them throughout physical memory with no significant explicit control over their self-interference potential. These experiments used such a small number of mbufs that this scattering did not cause an adverse effect. Improvements in cache effectiveness could be achieved by considering the layout of data in memory to minimize self-interference—uniform distribution over the cache is best with direct-mapped caches.

As context switches provide opportunities for active data to be flushed from the cache [11], access patterns should be optimized to improve temporal and spatial locality. Integrated layer processing [1] or application data units [3] could be used to accumulate data accesses in attempts to improve temporal and/or spatial locality. BSD's cache efficacy may be indicative of a light-weight kernel's small cache footprint when processing under light load conditions. Increasing the workload, simulating increases in incoming network bandwidth, was seen to decrease BSD's cache effectiveness. Operating systems with larger cache footprints, such as Mach, provide sufficient load internally that cache effectiveness is depressed even with no additional load.

Distributing operations among different user tasks induces the penalty of increased TLB and cache misses. Protection mechanisms on many common hardware architectures limit processes from making use of TLB and cache loads performed by other processes in other address spaces. Control of the placement and use of buffers through a mechanism such as fast buffers (fbufs) [5] could improve cache effectiveness by increasing sharing and decreasing TLB misses. Use of group TLBs could provide a reduction to a single miss. Distribution of tasks among components also likely increases the number of context switch points encountered along the processing path. The measurements made at copy-out time in these experiments were optimistic, but even more so for Mach; there are at least two context switch points which follow while data is transferred from the emulator's buffers to the user's buffers.[4]

Monolithic operating systems benefit from a smaller number of context switches in the network data processing path. The entire path is also contained within a single address space thereby decreasing the cost of cache and TLB loads induced by address space transitions. In order to increase cache effectiveness while using a microkernel operating system structure, the majority of network data processing could be migrated into the user's application. Implementing network protocols in the end-user application domain's (as opposed to separate network servers) [9, 16] would improve cache effectiveness by causing all network data accesses to occur with the user's address space. Cache interference is still a potential problem, but as many applications would be involved in network data processing the likelihood of self-interference would be decreased. Context switch points would be minimized to those required to process the network data, without those necessary to isolate the Unix server.

5 Conclusions

This paper reports on a set of experiments designed to evaluate the effectiveness of data caches in processing network I/O. Based on the results, we offer the following two broad conclusions.

First, the monolithic BSD kernel had significantly better cache and TLB behavior than the Mach microkernel. Although one reason for this result is that Mach has a bigger cache footprint than BSD, a second reason seems to be more intrinsic to conventional microkernel design—because Mach's network subsystem is implemented in a user-level network server, it is not able to exploit the TLB entries loaded by the kernel. On the other hand, once load was introduced, even BSD's cache efficacy begins to suffer.

Second, our results suggest three general rules for network subsystem design. The first is that because TLB usage has such an important impact, operating systems should be designed to take advantage of group TLB entries. A second is that because context switches have such an adverse effect on the cache, incoming

[4] Only one additional copy is required as the Unix server and emulator use shared memory to transfer data.

network data should be brought into the cache as late a possible (e.g., when first accessed by checksumming) and all accesses to network data should happen in the same context (e.g., in the application's protection domain). A third is that network buffers should be laid out contiguously and allocated in a LIFO manner, so as to minimize self-interference.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(1), 1993.

[2] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM 1990 Conference on Communications Architectures and Protocols*, September 1990.

[4] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network*, July 1993.

[5] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[6] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. Technical Report TR94-05, University of Arizona, Department of Computer Science, 1994.

[7] C. A. Gleason, L. Johnson, S. T. Mangelsdorf, T. O. Meyer, and M. A. Forsyth. VLSI circuits for low-end and midrange PA-RISC computers. *Hewlett-Packard Journal*, pages 12–22, August 1992.

[8] R. B. Lee. Precision architecture. *IEEE Computer*, pages 78–91, January 1989.

[9] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[10] H. E. Melesis and D. N. Serpanos. Designing communication subsystems for high-speed networks. *IEEE Network*, pages 40–46, July 1992.

[11] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Forth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[12] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Usenix 1990 Summer Conference*, pages 247–256, June 1990.

[13] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions on Networking*, pages 429–440, August 1993.

[14] K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February 1993.

[15] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[16] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[17] C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance atm host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240–253, February 1993.