Payload Caching: High-Speed Data Forwarding for Network Intermediaries

Ken Yocum and Jeff Chase
Department of Computer Science
Duke University
{grant,chase}@cs.duke.edu *

Abstract

Large-scale network services such as data delivery often incorporate new functions by interposing intermediaries on the network. Examples of forwarding intermediaries include firewalls, content routers, protocol converters, caching proxies, and multicast servers. With the move toward network storage, even static Web servers act as intermediaries to forward data from storage to clients.

This paper presents the design, implementation, and measured performance of *payload caching*, a technique for improving performance of host-based intermediaries. Our approach extends the functions of the network adapter to cache portions of the incoming packet stream, enabling the system to forward data directly from the cache. We prototyped payload caching in a programmable high-speed network adapter and a FreeBSD kernel. Experiments with TCP/IP traffic flows show that payload caching can improve forwarding performance by up to 60% in realistic scenarios.

1 Introduction

Data forwarding is increasingly common in large-scale network services. As network link speeds advance, networks are increasingly used to spread the functions of large servers across collections of networked systems, pushing functions such as storage into back-end networks. Moreover, systems for wide-area data delivery increasingly incorporate new functions — such as request routing, caching, and filtering — by "stacking" intermediaries in a pipeline fashion.

For example, a typical Web document may pass through a series of forwarding steps along the path from its home on a file server to some client, passing through a Web server and one or more proxy caches. Other examples of forwarding intermediaries include firewalls, content routers, protocol converters [10], network address translators (NAT), and "overcast" multicast nodes [13]. New forwarding intermediaries are introduced in the network storage domain [14, 2], Web services [12], and other networked data delivery.

This paper investigates a technique called *payload caching* to improve data forwarding performance on intermediaries. In this paper, we define *forwarding* as the simple updating of packet headers and optional inspection of data as it flows through an intermediary. Note that data forwarding is more general than packet forwarding. While it encompasses host-based routers, it also extends to a wider range of these intermediary services.

Payload caching is supported primarily by an enhanced network interface controller (NIC) and its driver, with modest additional kernel support in the network buffering and virtual memory system. The approach is for the NIC to cache portions of the incoming packet stream, most importantly the packet data payloads (as opposed to headers) to be forwarded. The host and the NIC coordinate use of the NIC's payload cache to reduce data transfers across the I/O bus. The benefit may be sufficient to allow host-based intermediaries where custom architectures were previously required. Section 2 explains in detail the assumptions and context for payload caching.

This paper makes the following contributions:

- It explores the assumptions underlying payload caching, and the conditions under which it delivers benefits. Quantitative results illustrate the basic properties of a payload cache.

*Author's address: Department of Computer Science, Duke University, Durham, NC 27708-0129 USA. This work is supported by the National Science Foundation (through EIA-9870724 and EIA-9972879), Intel Corporation, and Myricom.

- It presents an architecture and prototype implementation for payload caching in a programmable high-speed network interface, with extensions to a zero-copy networking framework [5] in a FreeBSD Unix kernel. This design shows how the host can manage the NIC's payload cache for maximum flexibility.

- It presents experimental results from the prototype showing forwarding performance under payload caching for a range of TCP/IP networking traffic. The TCP congestion control scheme adapts to deliver peak bandwidth from payload caching intermediaries.

- It outlines and evaluates an extension to payload caching, called *direct forwarding*, that improves forwarding performance further when intermediaries access only the protocol headers.

This paper is organized as follows. Section 2 gives an overview of payload caching and its assumptions. Section 3 outlines interfaces and extensions for payload caching at the boundary between a host and its NIC. Section 4 describes our payload caching prototype using Myrinet and FreeBSD. Section 5 examines the behavior and performance of payload caching. Section 6 describes related work and outlines future research. Section 7 concludes.

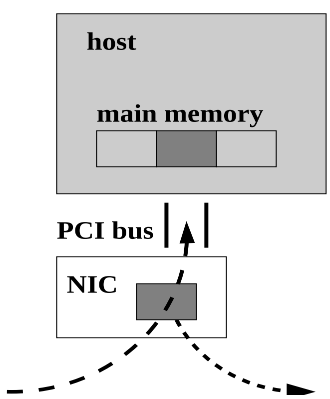

Figure 1: Forwarding a data payload with payload caching.

2 Overview

The payload caching technique optimizes network communication for forwarding intermediaries. Payload caching targets a typical host-based structure, in which the forwarding logic runs on a CPU whose memory is separated from the network interface. The NIC moves data to and from host memory using Direct Memory Access (DMA) across an I/O bus, such as PCI.

A forwarding intermediary receives a stream of packets from the network. Each packet DMAs across the I/O bus into one or more buffers in host memory. The network protocol stack inspects the headers and delivers the data to an application containing the intermediary logic, such as a firewall or caching proxy. The application may examine some of the data, and it may forward some or all of the data (the payload) to another destination without modifying it.

Figure 1 shows the potential benefit of payload caching in this scenario. Ordinarily, forwarded data payloads cross the I/O bus twice, once on input and once on output. Payload caching leaves incoming payloads in place in NIC buffers after delivering them to the host. If the host forwards the data unchanged, and if the forwarded data is still cached on the NIC, then the output transfer across the bus is unnecessary. This reduces the bandwidth demand of forwarding on the I/O bus and memory system, freeing these resources for other I/O or memory-intensive CPU activity. Payload caching can be especially effective for intermediaries that do I/O to other devices, such as disk-based Web proxy caches.

Payload caching imposes little or no runtime cost, but it yields a significant benefit under the following conditions.

- The intermediary forwards a large share of its incoming data without modifying it. This is often the case for intermediaries for Web delivery, including caching proxies, firewalls, content routers, multicast overlay nodes, and Web servers backed by network storage. Payload caching also naturally optimizes multicast transmits, such as mirrored writes to a network storage server or to a network memory cache [9].

- The payload cache on the NIC is large enough to retain incoming payloads in the cache until the host can process and forward them. In practice, the amount of buffering required depends on the incoming traffic rate, traffic burstiness, and the CPU cost to process forwarded data. One contribution of this work is to empirically determine the hit rates for various payload cache sizes for TCP/IP streams. Section 5.3 presents experimental results that show good hit rates at forwarding speeds up to

1 Gb/s and payload cache sizes up to 1.4 MB.

- Forwarded data exits the intermediary by the same network adapter that it arrived on. This allows the adapter to obtain the transmitted data from its payload cache instead of from the intermediary's memory. Note that this does not require that the output link is the same as the input link, since many recent networking products serve multiple links from the same adapter for redundancy or higher aggregate bandwidths. Payload caching provides a further motivation for multi-ported network adapters.
- The NIC supports the payload cache buffering policies and host interface outlined in Section 3. Our prototype uses a programmable Myrinet NIC, but the scheme generalizes easily to a full range of devices including Ethernet and VI NICs with sufficient memory.

While the payload caching idea is simple and intuitive, it introduces a number of issues for its design, implementation, and performance. How large must a payload cache be before it is effective? What is the division of function between the host and the NIC for managing a payload cache? How does payload caching affect other aspects of the networking subsystem? How does payload caching behave under the networking protocols and scenarios used in practice? The rest of this paper addresses these questions.

3 Design of Payload Caching

This section outlines the interface between the host and the NIC for payload caching, and its role in the flow of data through the networking subsystem.

The payload cache indexes a set of buffers residing in NIC memory. The NIC uses these memory buffers to stage data transfers between host memory and the network link. For example, the NIC handles an incoming packet by reading it from the network link into an internal buffer, then using DMA to transmit the packet to a buffer in host memory. All NICs have sufficient internal buffer memory to stage transfers; payload caching requires that the NIC contain sufficient buffer memory to also serve as a cache. For simplicity, this section supposes that each packet is cached in its entirety in a single host buffer and a single NIC buffer, and that the payload cache is fully effective even if the host forwards only portions of each packet unmodified. Section 4 fills in important details of host and NIC buffering left unspecified in this section.

The host and NIC coordinate use of the payload cache and cooperate to manage associations between payload cache entries and host buffers. A key goal of our design is to allow the host — rather than the NIC — to efficiently manage the placement and eviction in the NIC's payload cache. This simplifies the NIC and allows flexibility in caching policy for the host.

Figure 2 depicts the flow of buffer states and control through the host's networking subsystem. Figure 3 gives the corresponding state transitions for the payload cache. The rest of this section refers to these two figures to explain interactions between the host and the NIC for payload caching.

The dark horizontal bar at the top of Figure 2 represents the boundary between the NIC and the host. We are concerned with four basic operations that cross this boundary in a typical host/NIC interface. The host initiates *transmit* and *post receive* operations to send or receive packets. For example, the host network driver posts a receive by appending an operation descriptor to a NIC receive queue, specifying a host buffer to receive the data; the NIC delivers an incoming packet header and payload by initiating a DMA operation from NIC memory to the host buffer. In general, there are many outstanding receives at any given time, as the host driver attempts to provide the NIC with an adequate supply of host buffers to receive the incoming packet stream. When a transmit or receive operation completes, the NIC signals *receive* and *transmit complete* events to the host, to inform it that the NIC is finished filling or draining buffers for incoming or outgoing packets.

Payload caching extends these basic interactions to enable the host to name NIC buffers in its commands to the NIC. This allows the host to directly control the payload cache and to track NIC buffers that have valid cached images of host buffers. To avoid confusion between host memory buffers and internal NIC buffers, we refer to NIC buffers as payload cache *entries*. For the remainder of this paper, any use of the term *buffer* refers to a host memory buffer, unless otherwise specified.

Each payload cache entry is uniquely named by an *entry ID*. The host network driver specifies an entry ID of a NIC buffer to use for each host buffer in a newly posted transmit or receive. This allows the host to control which internal NIC buffers are used to stage transfers between host memory and the net-

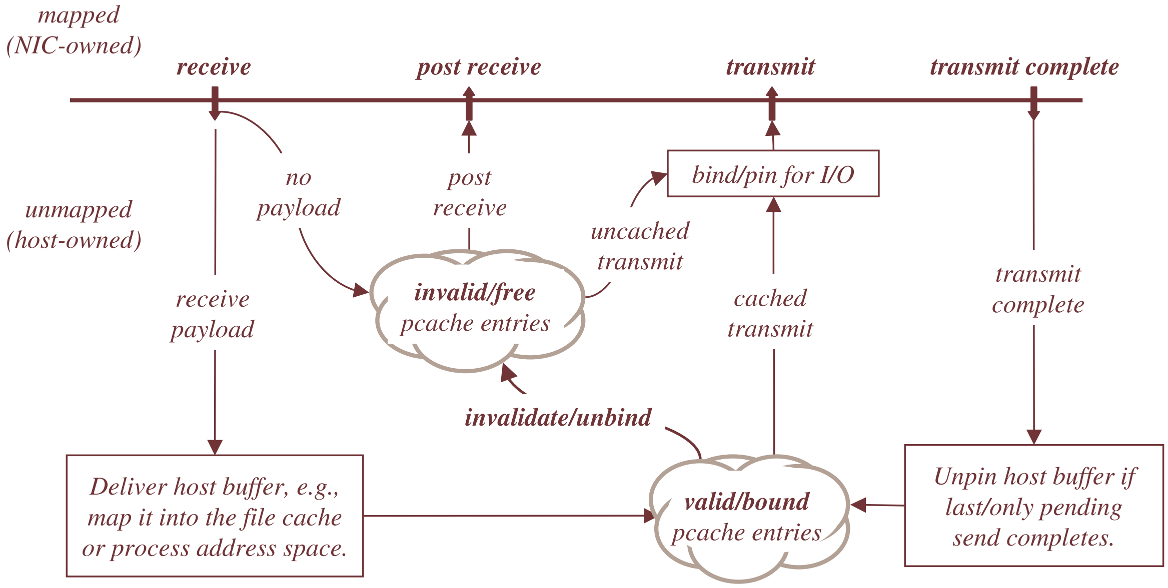

Figure 2: The flow of host buffers and payload cache entries through the networking subsystem.

work links. The NIC retains the data from each transfer in the corresponding entry until the host commands the NIC to reuse that entry for a subsequent transfer. Thus each transfer effectively loads new data into the payload cache; the host maintains an association between the host buffer and its payload cache entry as long as the entry's cached image of the buffer remains valid. If the host then initiates a subsequent transmit from the same buffer without modifying the data, the host sets a field in the descriptor informing the NIC that it may transmit data cached in the specified entry rather than fetching the data from the host buffer using DMA. This is a payload cache hit.

By specifying the entry ID for a transmit or receive, the host also controls eviction of data from the payload cache. This is because I/O through a payload cache entry may displace any data previously cached in the target entry. It is easy to see that most-recently-used (MRU) is the best replacement policy for the payload cache when the host forwards data in FIFO order. This is discussed further in Section 5.3.

We use the following terminology for the states of payload cache entries and host buffers. An entry is *valid* if it holds a correct copy of some host buffer, else it is *invalid*. A host buffer is *cached* if some valid entry holds a copy of it in the payload cache, else it is *uncached*. An entry is *bound* if it is associated with a buffer, else it is *free*. A buffer is *bound* if it is associated with an entry, else it is *unbound*. A bound (*buffer*,*entry*) pair is *pending* if the host has posted a transmit or receive operation to the NIC specifying that pair and the operation has not yet completed. Note that a bound buffer may be uncached if it is pending.

Initially, all entries are in the *free* state. The host driver maintains a pool of entry IDs for free payload cache entries, depicted by the cloud near the center of Figure 2. The driver draws from this pool of free entries to post new receives, and new transmits of uncached buffers. Before initiating the I/O, the operating system pins its buffers, binds them to the selected payload cache entries, and transitions the entries to the *pending* state. When the I/O completes, the NIC notifies the host with a corresponding *receive* or *transmit complete* notification via an interrupt. A *receive* may complete without depositing valid cacheable data into some buffer (e.g., if it is a short packet); in this case, the driver immediately unbinds the entry and returns it to the free pool. Otherwise, the operating system delivers the received data to the application and adds the bound $(buffer, entry)$ pair to its *bound entry* pool, represented by the cloud in the lower right of Figure 2.

On a transmit, the driver considers whether each buffer holding the data to be transmitted is bound to a valid payload cache entry. If the buffer is unbound, the driver selects a new payload cache entry from the free pool to stage the transfer from the buffer. If the buffer is already bound, this indicates that the host is transmitting from the same buffer used in a previous transmit or receive, e.g., to forward the payload data to another destination. This yields a payload cache hit if the associated entry is valid. The host reuses the payload cache entry for the transmit, and sets a field in the operation descriptor indicating that the entry is still valid.

After the transmit completes, the driver adds the entry and buffer pairing to the bound entry pool. Regardless of whether the transmit was a payload cache hit, the entry is now valid and bound to the host buffer used in the transmit. A subsequent transmit of the same data from the same buffer (e.g., as in a multicast) yields a payload cache hit.

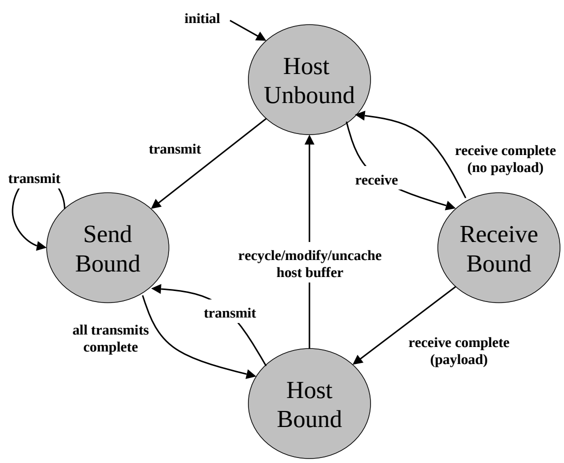

Figure 3: Payload cache entry states and transitions.

Figure 3 summarizes the states and transitions for payload cache entries. Initially, all entries are in the free state at the top of the figure. If the driver posts a transmit or a receive on an unbound/uncached host buffer, it selects a free NIC payload cache entry to bind to the buffer and stage the transfer between the network link and host memory. This causes the selected entry to transition to the left-hand **send-bound** state for a pending transmit, or to the right-hand **receive-bound** state for a pending receive.

In the **send-bound** and **receive-bound** states in Figure 3, the entry and buffer are bound with a pending I/O operation. For a transmit, the entry is marked valid as soon as the transfer initiates; this allows subsequent transmits from the same buffer (e.g., for a multicast) to hit in the payload cache, but it assumes that the NIC processes pending transmits in FIFO order. For a receive, the entry is marked valid only on completion of the received packet, and only if the received packet deposited cacheable data in the posted buffer (a short packet might not occupy all posted buffers).

A valid payload cache entry transitions to the bottom **host-bound** state when the pending transmit or receive completes. In this state, the entry retains its association with the host buffer, and caches a valid image of the buffer left by the completed I/O. Subsequent transmits from the buffer in this state lead back to **send-bound**, yielding a payload cache hit.

Once a binding is established between a host buffer and a valid payload cache entry (the **host-bound** state in Figure 3, and the bottom cloud in Figure 2), the operating system may break the binding and invalidate the payload cache entry. This returns the payload cache entry to the free pool, corresponding to the initial **host-unbound** state in Figure 3, or to the top cloud in Figure 2. This system must take this transition in the following cases:

- The system delivers the payload data to some application, which subsequently modifies the data, invalidating the associated payload cache entry.
- The system links the data buffer into the system file cache, and a process subsequently modifies it, e.g., using a *write* system call.
- The system releases the buffer and recycles the memory for some other purpose.
- The system determines that the cached entry is not useful, e.g., it does not intend to forward the data.
- There are no free payload cache entries, and the driver must evict a bound entry in order to post a new transmit or receive operation.

The payload cache module exports an interface to higher levels of the OS kernel to release or invalidate a cache entry for these cases. In all other respects payload caching is hidden in the NIC driver and is transparent to upper layers of the operating system.

4 Implementation

This section describes a prototype implementation of payload caching using Myrinet, a programmable high-speed network interface. It extends the design overview in the previous section with details relating to the operating system buffering policies.

Payload Cache Operations (exported to network driver)	
pc_receive_bind(physaddr)	Invalidate old binding if present, and bind the replacement payload cache entry with a host physical frame.
pc_send_bind(physaddr)	If the buffer is cached on the adapter, use existing binding, else find replacement and create new binding.
pc_receive_complete(physaddr)	The cache entry is now valid/bound.
pc_send_complete(physaddr)	If this is the last outstanding send, the cache entry is now valid/bound.
Payload Cache Management (exported to operating system)	
pc_invalidate_binding(physaddr)	Invalidate the payload cache entry bound to this physical address; the entry is now *host_unbound*.
pc_advise(physaddr, options)	Advise the payload cache manager to increase or decrease the payload cache entries priority.

Table 1: Payload Cache module APIs for the network driver and OS kernel.

We implemented payload caching as an extension to Trapeze [1, 4], a firmware program for Myrinet, and associated Trapeze driver software for the FreeBSD operating system. The host-side payload cache module is implemented by 1600 lines of new code alongside a Trapeze device support package below the driver itself. While our prototype implementation is Myrinet-specific, the payload caching idea applies to Gigabit Ethernet and other network interfaces.

Our prototype integrates payload caching with FreeBSD extensions for zero-copy TCP/IP networking [5]. This system uses page remapping to move the data between applications and the operating system kernel through the socket interface, avoiding data copying in many common cases. This allows us to explore the benefit of payload caching for intermediaries whose performance is not dominated by superfluous copying overhead. Copy avoidance also simplifies the payload cache implementation because forwarded data is transmitted from the same physical host buffer used to receive it. Thus there is at most one host buffer bound to each payload cache entry.

The Trapeze network interface supports page remapping for TCP/IP networking by separating protocol headers from data payloads, and depositing payloads in page-aligned host payload buffers allocated from a pool of VM page frames by the driver. The payload caching prototype manages a simple one-to-one mapping of bound payload cache entries with cached host memory page frames; the buffer bound to each payload cache entry is identified by a simple physical address.

Any modification to a cached buffer page in the host invalidates the associated payload cache entry, if any. Page protections may be used to trap buffer updates in user space. Note, however, that changes or reconstruction of packet headers does not invalidate the cache entries for the packet payload. For example, a Web server accessing files from an NFS file server and sending them out over an HTTP connection may use the payload cache effectively.

4.1 Payload Cache Module

Table 1 shows the interface exported by the payload cache module (*pcache*) to the Trapeze network driver and upper kernel layers. When the driver posts a transmit or receive, it invokes the *pc_receive_bind* or *pc_send_bind* routine in *pcache* to check the binding state of the target host buffer frames, and establish bindings to payload cache entries if necessary. The *pcache* module maintains a pcache entry table storing the physical address of the buffer bound to each entry, if any, and a *binding table* storing an entry ID for each frame of host memory. If a posted buffer frame is not yet bound to an entry, *pcache* finds a free entry or a suitable bound entry to evict.

When a send or receive completes, the driver invokes the *pcache* *pc_send_complete* or *pc_receive_complete* routine. If there are no more pending I/O operations on an entry, and the recently completed I/O left the entry with valid data, then *pcache* transitions the entry to the **host-bound** state, shown earlier in Figure 3.

The *pcache* module exports routines called *pc_invalidate_binding* and *pc_advise* to the upper layers of the operating system kernel. The kernel uses these to invalidate a payload cache entry when its bound buffer is modified, or to inform *pcache* that the cached data is or is not valuable. For example, the OS may call *pc_advise* to mark an entry as an eviction candidate if its payload will not be forwarded.

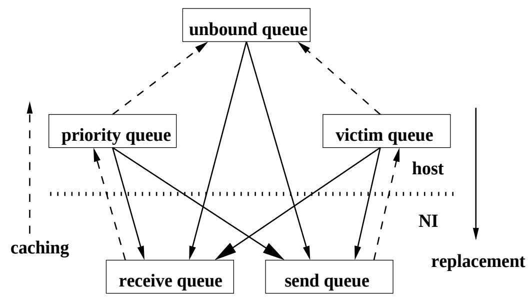

Figure 4: Queues for payload caching and replacement.

The prototype gives eviction priority to data that has been received but not yet transmitted. Any payload cache entry that is not pending resides on one of three replacement queues: unbound (free), priority, and victim. Figure 4 shows the movement of payload cache entries between these queues and the NIC send/receive queues. Entries for completed sends move to the victim queue, while entries for completed receives move to the priority queue. Entries on either victim or priority transition to the unbound queue if they are invalidated or demoted by *pc_advise*. An evicted entry may come from any of these queues, in the following preference order: unbound, victim, priority.

Note that *pcache* manages the payload cache entirely within the host, including invalidation and replacement. Support for payload caching on the NIC is trivial. The host piggybacks all payload cache directives on other commands to the NIC (transmit and post receive), so payload caching imposes no measurable device or I/O overhead.

4.2 Direct Forwarding

In normal forwarding operation, a payload caching host receives control and payload from the NIC, but transmits only headers across the I/O bus, sending forwarded payloads from the cache. For intermediaries that do not access most payloads — such as protocol translators, multicast nodes, or content switches — a natural progression is to extend the separation of control and payload data paths. In this case the NIC only passes control headers to the host, not data payloads. We term this configuration DIRECT forwarding (in contrast to PCACHE forwarding). Our prototype supports DIRECT forwarding mode with a small extension to the NIC firmware and a small change to the *pcache* module and driver. Payload cache entry management does not change. Experimental results in Section 5 show that DIRECT enables forwarding at link speeds, limited only by the CPU overhead for the forwarding logic. However, a pure DIRECT policy is appropriate only when the payload cache is adequately sized for the link or if the send rate is held below the level that overflows the cache. This is because evictions in a DIRECT payload cache discard the packet data, forcing the driver to drop any packet that misses in the payload cache in DIRECT mode.

Section 5.5 shows that TCP congestion control adapts to automatically deliver maximum allowable bandwidth through a DIRECT forwarder with very low miss rates in the presence of these packet drops. Even so, DIRECT is narrowly useful as implemented in our prototype. It would be possible to enhance its generality by extending the NIC to DMA direct-cached payloads to the host before eviction or on demand. Another alternative might be to extend the NIC to adaptively revert from DIRECT to PCACHE as it comes under load. We have not implemented these extensions in our prototype, but our implementation is sufficient to show the potential performance benefit from these more general approaches.

5 Payload Caching Performance

This section explores the effectiveness of the payload caching prototype for a simple kernel-based forwarding proxy. The results show the effect of payload caching on forwarding latency and bandwidth for TCP streams and UDP packet flows, varying the payload cache size, number of concurrent streams, packet size, and per-packet processing costs in the forwarding host CPU.

While the hit rate in the payload cache directly affects the increase in throughput and decrease in latency, it is not simply a function of cache size or replacement policy. Understanding the interplay between payload caching and forwarder behavior allows us to establish "real-world" performance under a variety of scenarios.

5.1 Experimental Setup

We ran all experiments using Dell PowerEdge 4400 systems on a Trapeze/Myrinet network. The Dell 4400 has a 733 MHz Intel Xeon CPU (32KB L1 cache, 256KB L2 cache), a ServerWorks ServerSet III LE chipset, and 2-way interleaved RAM. End systems use M2M-PCI64B Myrinet adapters with 66 MHz LANai-7 processors. The forwarder uses a more powerful pre-production prototype Myrinet 2000 NIC with a 132 MHz LANai-9 processor, which

Packet Size	Point to Point	Forwarding	PCACHE	DIRECT
1.5 KB	82.31μs	153.86μs	140.15μs	131.5μs
4 KB	108.36μs	224.68μs	191.68μs	173.71μs
8 KB	159.2μs	326.88μs	285.94μs	260.74μs

Table 2: One-way latency of UDP packets through an intermediary.

does not saturate at the forwarding bandwidths achieved with payload caching. The Myrinet 2000 NIC on the forwarder uses up to 1.4 MB of its onboard RAM as a payload cache in our experiments. All NICs, running Trapeze firmware enhanced for payload caching, are connected to PCI slots matched to the 1 Gb/s network speed. Since the links are bidirectional, the bus may constrain forwarding bandwidth.

All nodes run FreeBSD 4.0 kernels. The forwarding proxy software used in these experiments consists of a set of extensions to an IP firewall module in the FreeBSD network stack. The forwarder intercepts TCP traffic to a designated virtual IP address and port, and queues it for a kernel thread that relays the traffic for each connection to a selected end node. Note that the forwarder acts as an intermediary for the TCP connection between the end nodes, rather than maintaining separate connections to each end node. In particular, the forwarder does no high-level protocol processing for TCP or UDP other than basic header recognition and header rewriting to hide the identity of the endpoints from each other using Network Address Translation (NAT). This software provides a basic forwarding mechanism for an efficient host-based content switch or load-balancing cluster front end. It is equivalent to the kernel-based forwarding supported for application-level proxies by TCP splicing [8].

To generate network traffic through the forwarder we used *netperf* version 2.1pl3, a standard tool for benchmarking TCP/IP networking performance, and *Flowgen*, a network traffic generator from the DiRT project at UNC.

5.2 Latency

Table 2 gives the latency for one-way UDP transfers with packet sizes of 1500 bytes, 4KB, and 8KB. Interposing a forwarding intermediary imposes latency penalties ranging from 86% for 1500-byte packets to 105% for 8KB packets. Payload caching (PCACHE) reduces this latency penalty modestly, reducing forwarding latency by 8% for 1500-byte packets, 14% for 4KB packets, and 12% for 8KB packets. Direct forwarding (DIRECT) reduces forwarding latency further: the total latency improvement for DIRECT is 14% for 1500-byte packets, 22% for 4KB packets, and 20% for 8KB packets.

This experiment yields a payload cache hit for every forwarded packet, regardless of cache size. The resulting latency savings stems from reduced I/O bus crossings in the forwarder. PCACHE eliminates the I/O bus crossing on transmit, and DIRECT eliminates bus crossings on both transmit and receive. For all experiments the NIC uses a store-and-forward buffering policy, so I/O bus latencies are additive.

Propagation delays are higher in wide-area networks, so the relative latency penalty of a forwarding intermediary is lower. Therefore, the relative latency benefit from payload caching is also lower.

5.3 Payload Cache Size and Hit Rate

The next experiment explores the role of the forwarder's packet processing overhead on payload cache hit rates across a range of cache sizes. It yields insight into the amount of NIC memory needed to achieve good payload cache hit rates under various conditions.

The NIC deposits incoming packets into host memory as fast as the I/O bus and NIC resources allow, generating interrupts to notify the host CPU of packet arrival. In FreeBSD, the NIC driver's receiver interrupt handler directly invokes the IP input processing routine for all new incoming packets; this runs the protocol and places the received data on an input queue for delivery to an application. Incoming packets may accumulate on these input queues if the forwarder CPU cannot keep up with the incoming traffic, or if the incoming traffic is bursty. This is because the NIC may interrupt the application for service from the driver as more packets arrive.

The behavior of these queues largely determines the hit rates in the payload cache. Consider our simple forwarder example. The forwarder application runs as a thread within the kernel, and the network driver's incoming packet handler may interrupt it.

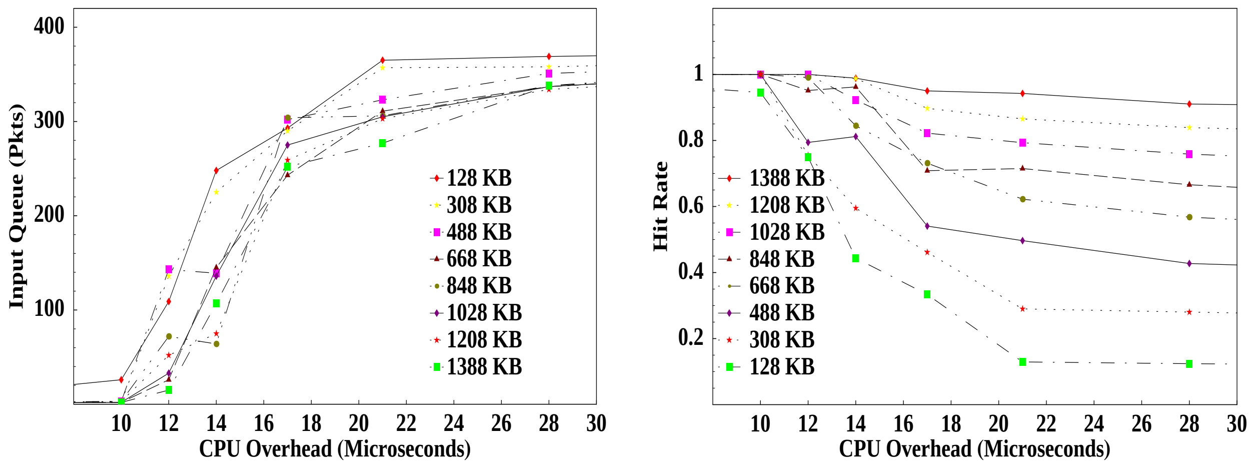

Figure 5: Forwarder queue size and hit rate for a 25 MB/s (200 Mb/s) stream of 4KB UDP packets. Each line shows results for a given effective payload cache size across a range of per-packet forwarding overheads on the x-axis.

As the driver consumes each incoming packet accepted by the NIC, it allocates a new host buffer and payload cache entry to post a new receive so the NIC may accept another incoming packet. Under ideal conditions the driver simply allocates from unused entries released as the forwarder consumes packets from its input queue and forwards their payloads. However, suppose traffic bursts or processing delays cause packets to accumulate on the input queue, awaiting forwarding. Since each buffered packet consumes space in the payload cache, this forces *pcache* to replace payload cache entries for packets that have not yet been forwarded, reducing the hit rate.

It is easy to see that under MRU replacement the payload cache hit rate for the forwarder will roughly equal the percentage of the buffered packet payloads that fit in the payload cache. Once an MRU cache is full, any new entries are immediately evicted. With FIFO forwarding, every buffered payload that found space in the cache on arrival ultimately yields one hit when it is forwarded; every other buffered payload ultimately yields one miss. Thus the average hit rate can be found by determining the average forwarder queue length — the number of payloads buffered for forwarding in the host — and dividing into the payload cache size. Note that MRU is the optimal behavior in this case because there can be no benefit to replacing an older cached payload with a newer one until the older one has been forwarded.

Queuing theory predicts these forwarder queue lengths under common conditions, as a function of forwarder CPU overhead (per-packet CPU service demand) or CPU utilization. This experiment illustrates this behavior empirically, and also shows an interesting feedback effect of payload caching. We added a configurable per-packet CPU demand to the forwarder thread, and measured forwarder queue lengths and payload cache hit rate under a 25 MB/s (200 Mb/s) load of 4KB UDP packets. We used the UNC *Flowgen* tool to generate poisson-distributed interarrival gaps for even queue behavior. This allows us to explore the basic relationship between CPU demand and payload cache hit rate without the "noise" of the burstier packet arrivals common in practice.

The left-hand graph of Figure 5 shows the average number of packets queued in the intermediary as a function of the CPU demand. We ran several experiments varying the *effective payload cache size*, the number of payload cache entries not reserved for pending receives. As expected, the queues grow rapidly as the CPU approaches saturation. In this case, the OS kernel bounds the input packet queue length at 400 packets; beyond this point the IP input routine drops incoming packets at the input queue. This figure also shows that the queues grow more slowly for large payload cache sizes. Queuing theory also predicts this effect: hits in the payload cache reduce the effective service demand for each packet by freeing up cycles in the I/O bus and memory system.

The right-hand graph of Figure 5 shows the average payload cache hit rate for the same runs. At low service demands, all packets buffered in the forwarder fit in the payload cache, yielding hit rates near 100%. As the forwarder queue lengths increase, a smaller share of the packet queue fits in the payload cache, and hit rates fall rapidly. As expected, the hit rate for each experiment is approximated by the portion of the packet queue that fits in the payload cache.

This experiment shows that a megabyte-range payload cache yields good hit rates if the CPU is powerful enough to handle its forwarding load without approaching saturation. As it turns out, this property is independent of bandwidth; if CPU power scales with link rates then payload caching will yield similar hit rates at much higher bandwidths. On the other hand, payload caching is not effective if the CPU runs close to its capacity, but this case is undesirable anyway because queueing delays in the intermediary impose high latency.

5.4 UDP Forwarding Bandwidth

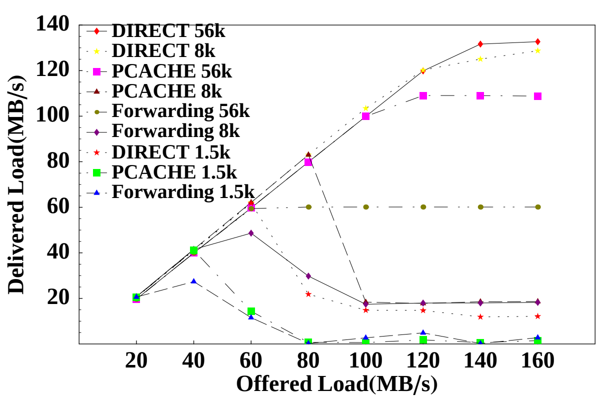

Figure 6: UDP forwarding bandwidth with PCACHE and DIRECT, for MTUs of 56KB, 8KB, and 1500 bytes.

The next experiment shows the bandwidth benefits of payload caching for *netperf* UDP packet flows. Figure 6 shows delivered UDP forwarding bandwidth as a function of input traffic rate for packet sizes of 1500 bytes, 8KB, and 56KB. The payload cache size is fixed at 1.4 MB. Bandwidth is measured as the number of bytes arriving at the receiver per unit of time. For these experiments, smaller packet sizes saturate the CPU at modest bandwidths; since UDP has no congestion control, the saturated intermediary drops many packets off of its input queues, but only after consuming resources to accept them. This livelock causes a significant drop in delivered bandwidth beyond saturation.

With 1500-byte packets, packet handling costs quickly saturate the forwarder's CPU, limiting forwarding bandwidth to 29 MB/s. PCACHE improves peak forwarding bandwidth by 35% to 40 MB/s. In this case, the benefit from PCACHE stems primarily from reduced memory bandwidth demand to forward each packet, as hits in the payload cache reduce the number of bytes transmitted from host memory.

The 8KB packets reduce packet handling costs per byte of data transferred, easing the load on the CPU. In this case, the CPU and the I/O bus are roughly in balance, and both are close to saturation at the peak forwarding rate of 48 MB/s. PCACHE improves the peak forwarding bandwidth by 75% to 84 MB/s due to reduced load on the I/O bus and on the forwarder's memory.

With 56KB packets, forwarding performance is limited only by the I/O bus. The base forwarding rate is roughly one-half the I/O bus bandwidth at 60 MB/s, since each payload crosses the bus twice. With PCACHE, the forwarding bandwidth doubles to 110 MB/s, approaching the full I/O bus bandwidth. This shows that payload caching yields the largest benefit when the I/O bus is the bottleneck resource, since it cuts bus utilization by half under ideal conditions. A faster CPU would show a similar result at smaller packet sizes. Note that for 56KB packets the forwarding rate never falls from its peak. This is because the CPU is not saturated; since the I/O bus is the bottleneck resource, the input rate at the forwarder is limited by link-level flow control on the Myrinet network. This is the only significant respect in which our experiments are not representative of more typical IP networking technologies.

The DIRECT policy reduces memory and I/O bus bandwidth demands further, and sustains much higher bandwidth across all packet sizes. At 1500 bytes, DIRECT reduces CPU load further than PCACHE, yielding a peak forwarding bandwidth of 60 MB/s. DIRECT can forward at a full 1 Gb/s for a 56KB packet size, with the CPU load at 20% and the I/O bus mostly idle for use by other devices. However, a payload cache miss affects bandwidth much more for DIRECT than for PCACHE, since a miss results in a packet drop. In these experiments, DIRECT suffered a 22% miss rate for a 100 MB/s input rate with 8KB MTU. The next section shows that TCP congestion control adapts to throttle the send rate when packets are dropped, yielding the best bandwidth and high hit rates for DIRECT.

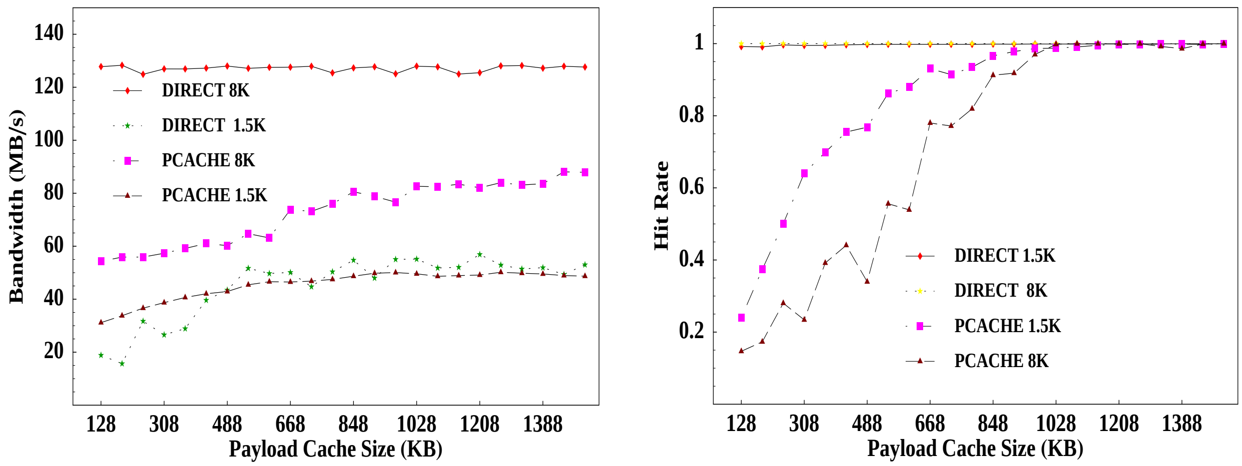

Figure 7: TCP bandwidth improvements with PCACHE and DIRECT for MTUs of 8KB and 1500 bytes, as a function of effective payload cache size up to 1.4 MB.

5.5 TCP Forwarding

We now show the effectiveness of payload caching for fast forwarding of TCP streams. For this experiment, we used *netperf* on four clients to initiate sixteen simultaneous TCP streams to a single server through a forwarding intermediary, with the interface Maximum Transmission Unit (MTU) configured to 1500 bytes or 8KB (Jumbo frames).

Figure 7 shows the resulting aggregate bandwidth and payload cache hit rate as a function of effective payload cache size. The graph does not show forwarding bandwidths without payload caching, since they are constant at the throughputs achieved with the smallest payload cache sizes. These base throughputs are 30 MB/s (240 Mb/s) with 1500-byte MTUs and 55 MB/s (440 Mb/s) with 8KB MTUs.

Using PCACHE, aggregate TCP bandwidth through the forwarder rises steadily as the payload cache size increases. With 1500-byte MTUs, payload caching improves bandwidth by 56% from 30 MB/s to a peak rate of 47 MB/s. With 8KB MTUs, payload caching improves bandwidth by 60% from 55 MB/s to 88 MB/s at the 1.4 MB payload cache size. It is interesting to note that these bandwidths are slightly higher than the peak bandwidths measured with UDP flows. This is because TCP's congestion control policy throttles the sender on packet drops. We measured slightly lower peak bandwidths for a single TCP stream; for example, a single stream with 8KB MTUs yields a peak bandwidth of 85 MB/s through a payload caching forwarder.

The right-hand graph in Figure 7 shows the payload cache hit rates for the same runs. Hit rates for the PCACHE runs rise steadily as the payload cache size increases, driving forwarding bandwidth up. For this experiment a megabyte of payload cache is sufficient to yield 100% hit ratios for all experiments.

Using direct forwarding (DIRECT) yields even higher peak bandwidths. A direct forwarder handles traffic at a full gigabit per second with 8KB MTUs, despite its I/O bus limitation. It might seem anomalous that bandwidth rises with larger cache sizes, even as the hit rate appears to be pegged at 100% even with small sizes. This effect occurs because all payload cache misses under DIRECT result in packet drops. Although a very small number of misses actually occur, they are sufficient to allow TCP's congestion control policy to quickly converge on the peak bandwidth achievable for a given cache size. With PCACHE, a payload cache miss only increases forwarding cost for an individual packet, which alone is not sufficient to cause TCP to throttle back until a queue overflows, forcing a packet drop. In all of our experiments, TCP congestion control policies automatically adjusted the send rate to induce peak performance from a payload caching forwarder.

6 Related Work

This section sets payload caching in context with complementary work sharing similar goals. Related techniques include peer-to-peer DMA, TCP splicing, and copy avoidance.

Like payload caching, peer-to-peer DMA is a technique that reduces data movement across the I/O bus for forwarding intermediaries. Data moves directly from the input device to the output device without indirecting through the host memory. Peer-to-peer DMA has been used to construct scalable host-based IP routers in the Atomic project at USC/ISI [18], the Suez router at SUNYSB [16], and Spine at UW [11]. The Spine project also explores transferring the forwarded payload directly from the ingress NIC to the egress NIC across an internal Myrinet interconnect in a scalable router. Like DIRECT payload caching, this avoids both I/O bus crossings on each NIC's host, reducing CPU load as well. In contrast to peer-to-peer DMA, payload caching assumes that the ingress link and the egress link share device buffers, i.e., they are the same link or they reside on the same NIC. While payload caching and peer-to-peer DMA both forward each payload with a single bus crossing, payload caching allows the host to examine the data and possibly modify the headers. Peer-to-peer DMA assumes that the host does not examine the data; if this is the case then DIRECT payload caching can eliminate all bus crossings.

TCP splicing [8] is used in user-level forwarding intermediaries such as TCP gateways, proxies [17], and host-based redirecting switches [6]. A TCP splice efficiently bridges separate TCP connections held by the intermediary to the data producer and consumer. Typically, the splicing forwarder performs minimal processing beyond the IP layer, and simply modifies the source, destination, sequence numbers, and checksum fields in each TCP header before forwarding it. A similar technique has also been used in content switches [3], in which the port controller performs the sequence number translation. Once a TCP splice is performed, the data movement is similar to the NAT forwarding intermediary used in our experiments.

The primary goal of TCP splicing is to avoid copying forwarded data within the host. Similarly, many other techniques reduce copy overhead for network communication (e.g., Fbufs [7], I/O-Lite [15], and other approaches surveyed in [5]). These techniques are complementary to payload caching, which is designed to reduce overhead from unnecessary I/O bus transfers.

7 Conclusion

Data in the Internet is often forwarded through intermediaries as it travels from server to client. As network speeds advance, the trend in Web architecture and other large-scale data delivery systems is towards increasing redirection through network intermediaries, including firewalls, protocol translators, caching proxies, redirecting switches, multicasting overlay networks, and servers backed by network storage.

Payload caching is a technique that reduces the overhead of data forwarding, while retaining the flexibility of host-based architectures for network intermediaries. By intelligently managing a cache of data payloads on the network adapter (NIC), the host can improve forwarding bandwidth and latency.

This paper shows how to incorporate payload caching into Unix-based frameworks for high-speed TCP/IP networking. It shows the interface between the host and the NIC and the new host functions to manage the payload cache. A key feature of our system is that the host controls all aspects of payload cache management and replacement, simplifying the NIC and allowing the host to use application knowledge to derive the best benefit from the cache. The NIC support for our payload caching architecture is straightforward, and we hope that future commercial NICs will support it.

Experimental results from the prototype show that payload caching and the direct forwarding extension improve forwarding bandwidth through host-based intermediaries by 40% to 60% under realistic scenarios, or up to 100% under ideal conditions. TCP congestion control automatically induces peak forwarding bandwidth from payload caching intermediaries. These bandwidth improvements were measured using effective payload cache sizes in the one-megabyte range on a gigabit-per-second network.

8 Availability

For more information please visit the website at `http://www.cs.duke.edu/ari/trapeze`.

9 Acknowledgments

Over the years many people have contributed to the development and success of the Trapeze project. Most notably Andrew Gallatin for his FreeBSD expertise and Bob Felderman at Myricom for his timely help. We thank the anonymous reviewers and our shepherd, Mohit Aron, for helpful critiques and suggestions.

References

[1] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *1998 Usenix Technical Conference*, June 1998.

[2] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[3] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proceedings of IEEE Infocom 2000*, March 2000.

[4] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network I/O with Trapeze. In *1999 Hot Interconnects Symposium*, August 1999.

[5] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications, Special Issue on High-Speed TCP*, 39(4):68–74, April 2001.

[6] A. Cohen, S. Rangarajan, and H. Slye. The performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

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

[8] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *USENIX Conference*, pages 327–334, January 1993.

[9] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[10] M. E. Fiuczynski, V. K. Lam, and B. N. Bershad. The design and implementation of an IPv6/IPv4 network address and protocol translator. In *Proceedings of USENIX*, 1998.

[11] Marc E. Fiuczynski, Brian N. Bershad, Richard P. Martin, and David E. Culler. SPINE: An operating system for intelligent network adapters. Technical Report UW TR-98-08-01, Washington University, Department of Computer Science, September 1998.

[12] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-based scalable network services. In *Proceedings of Symposium on Operating Systems Principles (SOSP-16)*, October 1997.

[13] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[14] David F. Nagle, Gregory R. Ganger, Jeff Butler, Garth Gibson, and Chris Sabol. Network support for network-attached storage. In *Proceedings of Hot Interconnects*, August 1999.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[16] P. Pradhan and T. Chiueh. A cluster-based, scalable edge router architecture. In *Proceedings of the 1st Myrinet Users Group Conference*, 2000.

[17] O. Spatscheck, J. Hansen, J. Hartman, and L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 2(8):146–157, 2000.

[18] S. Walton, A. Hutton, and J. Touch. Efficient high-speed data paths for IP forwarding using host based routers. In *Proceedings of the 9th IEEE Workshop on Local and Metropolitan Area Networks*, pages 46–52, November 1998.