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.

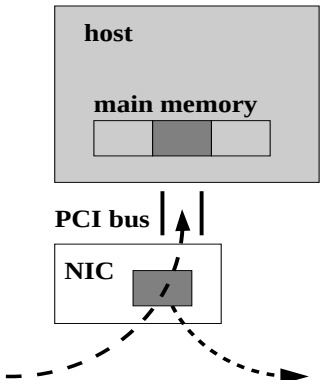

Figure 1: Forwarding a data payload with payload caching.

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-

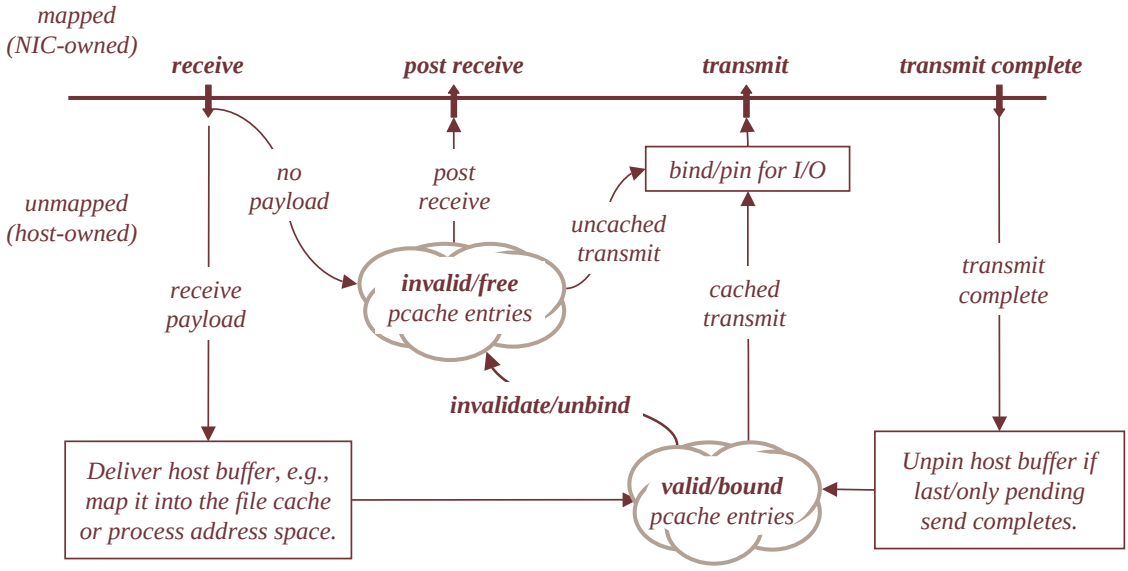

Figure 2: The flow of host buffers and payload cache entries through the networking subsystem.

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.

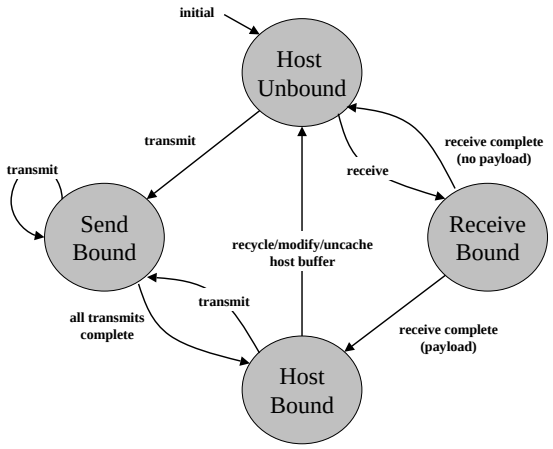

Figure 3: Payload cache entry states and transitions.

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.