Experiences with Fast Forwarding on Myrinet *

Kenneth G. Yocum, Jeffrey S. Chase
Department of Computer Science
Duke University
Durham, NC 27708-0129
{*grant, chase*}*@cs.duke.edu*
{*http://www.cs.duke.edu/ari/trapeze)*

Abstract

As cluster services become increasingly complex, awareness of the capabilities of those services are pushed back into the network, and service aware networking components are now common. Traffic entering and exiting cluster services is likely to be forwarded through a front-end as the performance and economic benefits of resource-allocation and load-balancing across the cluster service are realized. Packet routing, an end-to-end distributed service, is diverging from packet forwarding, a localized service specific task, as web serving, caching, redirection, and other general front-end host-based architectures increase in popularity.

This paper introduces payload caching, *a technique that caches incoming data payloads on the network interface, transparently reducing the number of I/O transfers needed to forward a packet. With payload caching, data is still transferred to host memory for intelligent redirection or host processing. Unmodified data remains cached on the network interface card(NIC) where it may be sent without crossing the I/O interconnect again. We report user-level results for three generations of LANai processor architecture and a queuing model showing that current NIC memory sizes are sufficiently large to get a 100% payload cache hit rate under certain conditions. Our prototype on the next generation Myrinet adapter (LANai-9) delivers an 89% improvement in forwarding bandwidth on a 32-bit 33Mhz PCI system, in this case bandwidth for a single 4KB packet stream goes from 56.67MB/s without payload caching to 106.43MB/s.*

*This work is supported by the National Science Foundation (EIA-9870724 and EIA-9972879), Intel Corporation, and Myricom.

1 Introduction

Packet forwarding is present in web redirectors, proxies, servers, or any front-end directing traffic to a back-end service [3, 5, 7]. These forwarding agents tend to inspect more deeply into the packet than do routers, which are typically concerned with the IP header. Host-based cluster front-ends [8, 1] enable application specific forwarding functionality [9], but can be limited by the PCI bus in common commodity host architectures. Payload caching promises to reduce the performance penalty associated with host-based forwarders.

Because the NIC orchestrates I/O data transfers, an obvious application for intelligent NICs is to manage and reduce the I/O load for network traffic. When the bus and network speeds are identical potential bandwidth can be halved using host-based front-ends, because the data makes two trips across the I/O bus (upon reception and again on transmission). While prior networking interfaces buffer a handful of packets, capable adapters can manage a payload cache to reduce host I/O load on commodity hardware for packet forwarding.

A cached payload forwarded directly from the interface avoids an I/O transfer to the NIC. Though payload caching does not apply to routers or firewalls because packets enter and exit the same NIC, it does reduce the load on the I/O interconnect, the CPU, and the memory bus for general purpose host-based front-ends. Peer-to-peer DMA is a complementary technique that has been used to improve host-based firewalls and routers [10] by selecting a peer NIC on the PCI bus as the DMA transaction destination, instead of host memory. This also halves the number of I/O transfers needed to forward a packet, but may place the onus of packet inspection and forwarding on the NIC's CPU resources [4].

Myrinet and Gigabit Ethernet adaptors must be capable of transferring data at gigabit rates while simultaneously performing extra computation to forward packets ef-

fectively from a payload cache. In our experiences with Trapeze [2], executing extra code on the LANai-4 was clearly making a tradeoff between performance and functionality. The removal or rearrangement of a few lines of Trapeze firmware code often resulted in bandwidth changes from five to ten percent. The current generation of 32-bit LANai-7 is clocked at 66Mhz with 2MB of memory on board; the next generation of LANai increases processing capacity, clocking a prototype LANai-9 at 133Mhz with 4MB of memory. Alteon Gigabit Ethernet adapters contain two 32-bit MIPS R4k processors running at 100 Mhz with 1 MB of memory. These network interface architectures present an environment in which it is feasible to execute extra code while transferring data at link speeds.

Three observations recommend using payload caching:

- Industry movements such as Infiniband may permanently marry network link rates to I/O interconnect speeds by using the same technology.
- Data that once was ignored in a router is commonly inspected for higher level load balancing or resource allocation algorithms in a forwarding agent.
- Recent gigabit network adapters have enough on-board *memory*, memory bandwidth, and CPU cycles to make payload caching effective at gigabit data rates.

This paper asks whether payload caching on the NIC is an important and feasible optimization to host-based forwarding architectures. We develop basic queuing models in order to answer how much cache space is needed on the network interface to support gigabit rates. We implement a user-level forwarding agent that uses Trapeze directly in order to support the basic queuing models with empirical data. We observe small numbers of collisions in the payload cache using an arbitrary indexing scheme to manage the replacement of cache entries on the NIC.

Section 2 examines our queuing models to discover the relative bottlenecks and queue sizes required for the different LANai adapters. Section 3 outlines the firmware implementation of payload caching in Trapeze. Next, Section 4 describes our user-level performance results for the three generation of LANai. Finally, we conclude in Section 5.

2 A Queuing Model for Payload Caching

For payload caching to be effective, there must be sufficient processing power and memory bandwidth on the interface to support bi-directional data streams at link rates. Forwarding packets is a special case of bi-directional traffic on a single interface; payload caching reduces the amount of data that flows through the NIC. This section describes a network queuing model for payload caching in order to ascertain both the system bottlenecks and cache size. Results

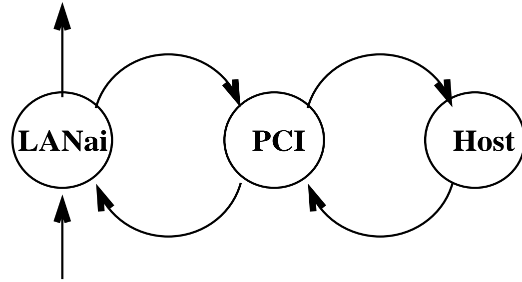

Figure 1. This queuing network is composed of three service centers: the NIC, PCI bus, and host.

from a user-level microbenchmark validate the assumptions and design of our model.

2.1 Model

Figure 1 shows an open queuing model with one job class, a packet. The visit count(V_k) for the NIC and host is one. Though the packet is actually touched twice by the NIC, it is easiest to collapse the interactions between sending and receiving into one service requirement per packet. The host need only service a packet once for forwarding. However the PCI bus has a visit count of two ($V_{bus} = 2$), once to deliver the data to the host and again to transfer the data back to the NIC. The purpose of payload caching is to reduce V_{bus} to one.

2.2 Parameterization and Bottleneck Analysis

In this section we parameterize the model with respect to three generations of Myrinet hardware, summarized in Table 1, and discover the bottleneck service centers. We conservatively assume that current host processors can service individual packets once every 20μs, or 50,000 packets a second. On our platforms interrupts are serviced in approximately 5μs, leaving 15μs for host processing. However, the Myrinet NIC is complicated to parameterize due to the multiple DMAs that can be in progress to LANai memory.

Memory clock rate and access arbitration rules influence the service requirement for the Myrinet NIC. The LANai memory is clocked to twice the NIC CPU speed; there are two memory cycles for every NIC processor cycle. The processor needs two memory cycles for a data and instruction fetch. The three on-board DMA engines may occupy one out of the two memory cycles per CPU cycle time. Finally, access to LANai memory is arbitrated in the following priority: external DMA, network receive DMA, net-

NIC	memory width & clock speed	processor speed	memory size
LANai 4	32 bits :: 66 Mhz	33 Mhz	1 MB
LANai 7	64 bits :: 132 Mhz	66 Mhz	2 MB
LANai 9	64 bits :: 266 Mhz	133 Mhz	4 MB

Table 1. Three versions of LANai architecture.

work send DMA, and LANai processor. If more than one DMA is scheduled the LANai processor may starve. For bi-directional streams or when forwarding packets, the memory is accessed by all DMA engines and the processor.

The model views the Myrinet NIC as one block of memory that requires x accesses in order to receive and forward a packet. We model the system with 4KB packets, the page size on our Intel systems. When forwarding data without payload caching, it takes four transfers of 4096 bytes or $(4 * 1024)$ 32-bit word accesses to LANai memory. The Trapeze Myrinet firmware executes approximately 550 instructions to forward one packet. Because the LANai-4 memory bandwidth is equal to the bus and link speed, it is the simplest Myrinet NIC to parameterize. On a LANai-4 each memory access takes 0.0152μs to complete; recalling that instructions need two memory accesses, the service requirement per forwarded packet on the LANai-4 is $(4096 + 1100) * 0.0152\mu s = 78.72\mu s$.

The same technique is used for the payload caching NIC service requirement, but with the payload transfer to the NIC removed. Figure 2 shows how LANai-4 memory cycles are split between Net-to-Card DMA (A1,B1,C1), Card-to-Host DMA (A2, B2), Card-to-Net DMA (A3,B3), and NIC packet processing. In this pipeline each operation takes approximately the same amount of time; it illustrates that LANai-4 memory behavior forces a forwarded packet to take two stages to complete. This is because both memory cycles are fully occupied and no further pipelining is available. Figure 3 shows the region in the dotted box in greater detail.

The NIC service requirement numbers must take into account the slower rate of the 32-bit 33 Mhz PCI bus, whose delay is illustrated here taking 31.03μs in the second stage, Card-to-Host. Figure 3 shows that the LANai requires 31.03μs to push data through the second stage. The LANai-4 takes just as long in the first stage as in the second, and its packet processing must be executed later. Though memory bandwidth on the LANai-4 is the bottleneck to increasing performance, limiting improvements on this NIC to 63.63MB/s, the host's I/O load is still halved.

Parameterizing the LANai-7 and LANai-9 is more challenging. Because they both have excess memory bandwidth, the CPU is not starved, and its cycles can proceed in parallel with one of the DMA accesses. However, the 32-bit 33 Mhz PCI bus still defines the latency of a 4KB Card-to-Host transfer and the link, running at 160MB/s, defines the

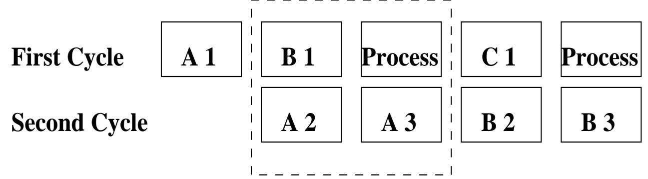

Figure 2. LANai memory has two cycles for each processor cycle. Each DMA engine occupies at most one memory cycle per processor cycle. This diagram illustrates how the memory cycles on the LANai-4 may be utilized when forwarding packets using data cached on the NIC. The area in the dotted box is inspected in greater detail in Figure 3.

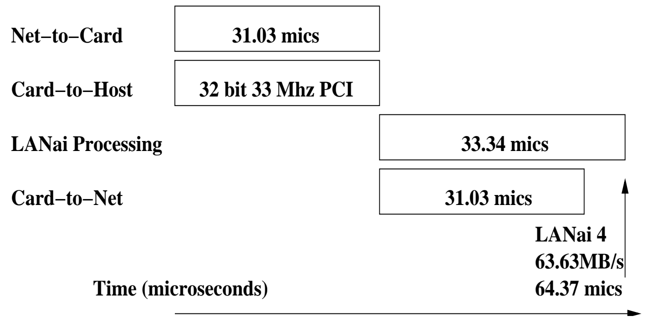

Figure 3. Illustrating the LANai memory priorities with a pipeline diagram of four stages representing a packet being forwarded through an adapter. The length of the blocks represents the time spent in each stage.

NIC	payload caching	D_{NIC}	$D_{PCI} = V_{PCI} * S_{PCI}$	$\lambda_{sat} * 4KB$	$\lambda_{seen} * 4KB$
LANai 4	OFF	$\mathbf{78.72\mu s}$	$62.06\mu s = 2 * 31.03$	52.03MB/s	49.89MB/s
LANai 7	OFF	$\mathbf{62.06\mu s}$	$\mathbf{62.06\mu s = 2 * 31.03}$	66MB/s	52.69MB/s
LANai 9	OFF	$\mathbf{62.06\mu s}$	$\mathbf{62.06\mu s = 2 * 31.03}$	66MB/s	54.23MB/s
LANai 4	ON	$\mathbf{64.37\mu s}$	$31.03\mu s = 1 * 31.03$	63.63MB/s	62.94MB/s
LANai 7	ON	$\mathbf{42.26\mu s}$	$31.03\mu s = 1 * 31.03$	96.92MB/s	89.10MB/s
LANai 9	ON	$\mathbf{34.28\mu s}$	$31.03\mu s = 1 * 31.03$	119.48MB/s	106.43MB/s

Table 2. Service requirements(S_x), visit counts(V_x), and service demand(D_x) for four different configurations. The model's parameters assume a 32-bit 33Mhz PCI bus, 4096 byte data payloads, and a host service requirement of 20μs. The bottleneck is emphasized in each case.

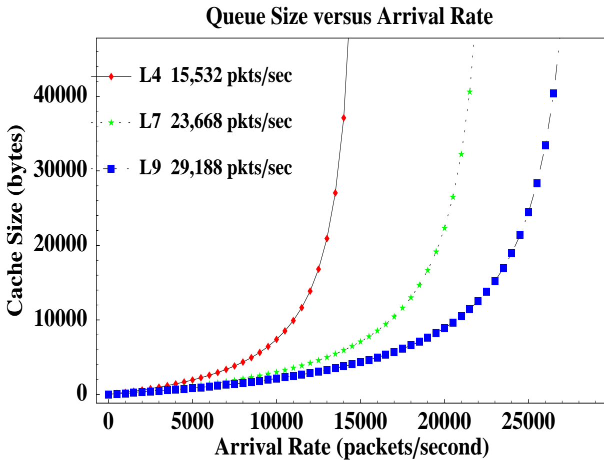

Figure 4. Payload cache sizes needed to sustain a 100% hit rate as the saturation arrival rates change per LANai version.

latency for Card-to-Net and Net-to-Card transfers. taking these two factors into consideration limits further performance gains to 96.92MB/s and 119.48MB/s for 4KB packets on the LANai-7 and LANai-9 respectively. Table 2 lists the resulting service times for all three LANai generations.

This table also lists λ_{sat}, the arrival rate at saturation of the bottleneck service center, and λ_{seen}, the observed maximum arrival rate. It is interesting to note that without the LANai-7, payload caching would have no benefit on bandwidth (though still reducing PCI and memory bus traffic) due to the amount of memory contention on the LANai-4. It is our belief that modern gigabit network adapters will not be the bottleneck in terms of internal memory bandwidths and CPU resources for payload caching.

2.3 Payload Cache Size

In order for NIC based payload caches to be effective, the cache must be large enough to handle traffic bursts. The previous section delineates the bottlenecks in our system as payload caching and faster NICs are introduced. The analysis bounds the processing time available to the host to sustain, at saturation, a 100% hit rate in the payload cache. It assumes a constant arrival rate, where the amount of buffering needed is small, simply enough to fill the pipeline. This is an upper bound on the benefit that payload caching gives to any system. However, it does not address variable network arrival rates, in other words, a workload more indicative of real traffic patterns.

First we use our model to determine the arrival rates at which the bottleneck service center is saturated, λ_{sat}, from our parameterization for the three LANai versions. We then use Little's Law to determine queue length which we multiply by 4KB to determine payload cache size. Figure 4 shows payload cache size for each LANai increasing as the arrival rate increases to saturation. Naturally, at saturation queue length becomes unbounded as does the size of the payload cache.

If the payload cache is managed as a FIFO, the queue length indicates how large the payload cache must be to sustain the arrival rate. This figure shows that the theoretical minimum amount of buffering required on the NIC for any LANai version at saturation is less than 64KB. Though this figure shows results only for 4KB packets, increasing packet size would have a negligible effect on queue size. This is because an increase in packet size would directly increase the bottleneck service requirement, namely the time of transfer across the 32-bit PCI bus. A doubling in service requirements means that λ_{sat} will halve; thus, bandwidth and the amount of buffering required remains constant.

Figure 4 also illustrates the limits at which host processing will bottleneck the system. If the host fails to forward a packet in the bottleneck service requirement (e.g., $42.25\mu s$ for the LANai-7), the queue grows quickly and the payload

Valid Bit	I/O Pending	Buffer State
$= 0$	$= 0$	Free
$= 0$	≥ 1	Fill
$= 1$	$= 0$	Valid
$= 1$	≥ 1	Drain

Table 3. States of buffers in the NIBC.

cache may begin to reclaim buffers that have yet to be forwarded. The payload cache must be large enough to handle bursts of activity, and well managed to allow buffers to live as long as possible in the cache to be forwarded. This study shows that a reasonably sized payload cache can be effective in reducing PCI load and increasing bandwidth when the NIC's memory bandwidth is sufficient. The next section describes our implementation and management system for a payload cache.

3 Implementation

Our payload cache implementation in Trapeze consists of approximately 550 instructions in the firmware and slight modifications to Trapeze host software. The payload cache is almost transparent to a forwarding agent, who is only required to assert entries in the cache have not been modified.

3.1 The Trapeze Payload Cache

Trapeze sends fixed-size control messages with optional attached payloads [11]; with the payload clearly separated from the control message, the payload cache implementation is simple. The control information is received and sent normally; packet headers fit in the control message and are easily modified. The control message's attached payload is cached on the NIC. The Trapeze firmware incorporates two rings, one for sending and one for receiving, strictly managed as circular producer/consumer queues with the host. Payload caching requires two main entry points in the firmware where ring entries are obtained for sending and receiving.

The payload cache is simply an array of host page size buffers in NIC memory. When the firmware finds a send ring entry ready to transmit, the firmware designates a payload cache buffer to the ring entry. Likewise, upon receiving a packet, the firmware designates a payload cache buffer to the receive ring entry.

A payload cache buffer can be in one of four states shown in Table 3, defined by a valid state bit and an I/O pending count. The fill state indicates that the buffer is being filled from the network on a receive or from the host on a send. Similarly, a buffer in the drain state is being sent to the network or received into host memory. Each buffer is also tagged with the host's physical DMA address in order to avoid sending invalid data.

The firmware increments the I/O pending count by two when a buffer is attached. On a send, it decrements the I/O pending count when the transfer from the host completes and again when the transfer to the net finishes. On a receive, the firmware decrements the I/O pending count as the transfers from the network and to the host complete. The valid bit is set after the first transfer is finished.

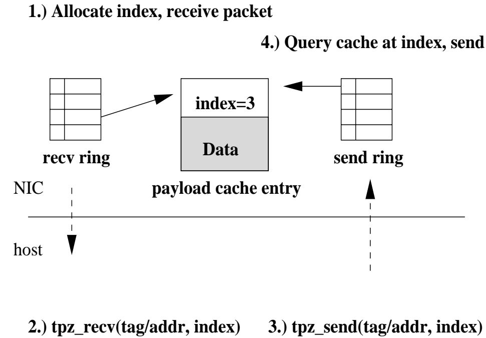

Figure 5. Steps to forward a packet: 1.) the firmware allocates a payload cache entry and receives the payload onto the NIC, 2.) the host software stores the binding between the host's physical DMA address and payload cache index, 3.) the host software uses that binding for each send from the same host address, and 4.) the firmware ensures the data is valid and the tag matches.

The four steps in forwarding a cached payload are illustrated in Figure 5. The Trapeze host software binds the payload cache index, simply an index into the array of payload cache buffers on the NIC, with the host's DMA address upon message reception. When sending from that same host DMA address, the host software recalls the binding and may notify the firmware either to use or invalidate the cached copy.

If the host decides to use the cached copy, the payload cache buffer's tag is first checked against the DMA address in the send ring entry. If the payload cache buffer on the NIC is either in the valid or drain state and the tag matches it can be immediately sent out on the wire. If the tags do not match, the operation must wait until the buffer is valid or free. However if the application notifies the firmware to invalidate this payload cache entry, then the firmware invalidates the current buffer, as long as it is valid, representing that host DMA address and initiates a DMA from host memory to the NIC. The firmware also implements the replacement policy of the payload cache.

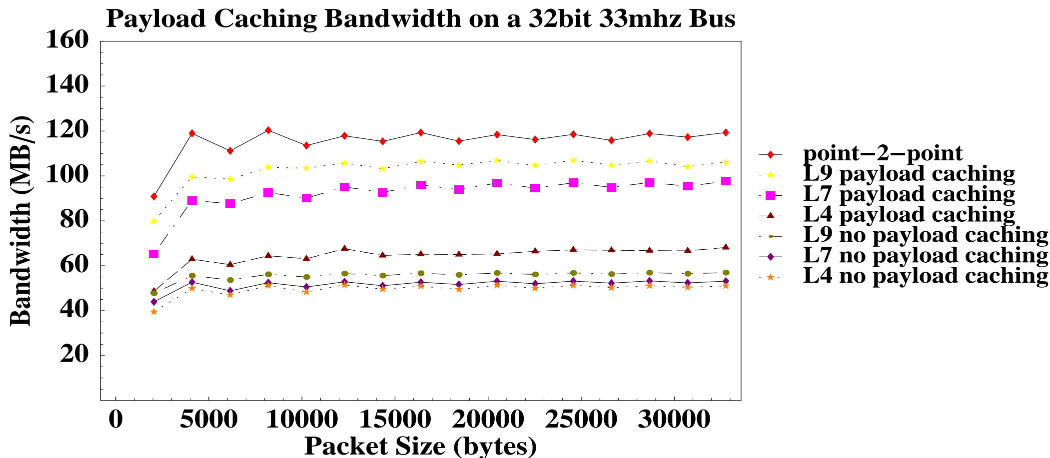

Figure 6. User-level forwarding bandwidth with three versions of Myrinet NIC on a 32-bit 33 Mhz PCI bus.

3.2 Arbitrary Payload Cache Indexing

In order to reduce misses, the payload cache should keep buffers that have not been forwarded as long as possible. The natural replacement policy for this behavior is FIFO; the replacement buffer belongs to the message that was received farthest in the past.

The Trapeze firmware simply allocates payload cache buffers in a circular fashion from the cache on a send or receive. If a buffer is allocated to the send ring, though previously bound to a receive ring entry, then the Trapeze firmware arbitrates access. Either operation will wait for the buffer to be valid with no I/O pending, since the tags do not match. A host's physical DMA address is never on the send ring and receive ring simultaneously. Naturally, a host that sends a stream from host memory may pollute the payload cache with traffic that is never re-sent. A concurrent forwarding stream may incur numerous payload cache misses, and benefit from the payload cache is reduced.

Host send pollution is prevented by the firmware allocating payload cache entries from a small send segment, typically four payload cache entries, when instructed to send from a previously uncached host physical DMA address. Similar arrangements can be made to ensure quality of service for streams that exhibit certain characteristics. For instance, the payload cache will not be as effective for streams of small packets. The firmware could easily re-use payload cache buffers for small packets that would benefit less from reducing I/O transfers, packets whose data is less important, or packets that will not be forwarded.

Though theoretically the payload cache does not need more than 64KB of buffer space, because Trapeze uses producer/consumer rings, the payload cache must be big enough to buffer all posted sends and unreceived messages. The size of the payload cache is therefore $\geq 2 * ringsize$. The megabyte of memory on the LANai-4 supports 128 4KB buffer entries with $ringsize = 64$, while the LANai-7 and LANai-9 have 384 4KB buffer entries and the same ring size.

4 Performance

We obtained these results using a user-level forwarding agent. The end hosts were 733 Mhz Pentium IIIs with a LANai-7 in a 64-bit 66 Mhz PCI slot. We used a 32-bit PCI with a 433 Mhz Pentium III to make the PCI bus the bottleneck service center. Our forwarding agent is single-threaded, and directly accesses the Trapeze interface with no operating system intervention, i.e., it uses polling to receive packets. Packets are forwarded as soon as they are received. This results in a modest $4\mu s$ service time per packet.

We observe a 100% hit rate in the payload cache for all tests, influenced by the synchronous behavior of Trapeze's producer/consumer rings, link level flow control, and the FIFO payload cache replacement policy. As long as the number of payload cache entries is at least twice the ring size, the forwarding NIC will exert back pressure to the sender if the forwarding agent stalls longer than the bottleneck service requirement. The Trapeze firmware cannot receive without a receive ring entry and, therefore, will not evict packets which have yet to be forwarded. These microbenchmarks therefore illustrate the performance of a payload cache in which every forwarded payload hits in the cache, e.g., no payload must make a second trip across the

PCI bus.

4.1 User-Level Forwarding

As described in Section 2, the 32-bit PCI bus is the bottleneck when forwarding packets. Figure 6 shows the effects of payload caching on the LANai-4, 7, and 9 for streams of packets ranging in size from 2KB to 32KB. The slight saw tooth pattern is evidence of Trapeze's use of scatter/gather, where an additional transfer is required when the packet size is not evenly divisible by host page size.

The top line represents point-to-point bandwidth between two LANai-7's. The bottom line is the forwarding bandwidth without payload caching on a LANAi-4; as our model predicts, bandwidth does not exceed 52MB/s. While increased NIC memory bandwidth and CPU speed alleviate internal NIC bottlenecks present in the LANai-4, neither the LANai-7 or LANai-9 significantly improve non-payload cache performance while the bus is the bottleneck.

However, as soon as we remove this bottleneck by using a payload cache, we see dramatic jumps in forwarding bandwidth. The three lines for forwarding with payload caching show performance correlates directly with improved LANai processor and memory performance. The LANai-4 remains bottlenecked at its memory bandwidth, while the LANai-7 and LANai-9 experience bottlenecks due to data transfers on the 32-bit PCI bus. The faster processor on the LANai-9 improves performance over the LANai-7 by about 10%, forwarding at 106.43MB/s and attaining an 89% improvement over results without payload caching.

Payload caching will not have an effect on end-to-end forwarding bandwidth when the host's I/O interconnect is not the bottleneck. Figure 7 shows the same forwarding tests on a 64-bit 66 Mhz PCI bus. In this case the forwarding bandwidth for both the LANai-7 and LANai-9 is unchanged by reducing the number of bus crossings; it is sustained at the same rate as point-to-point bandwidth. Even though the end points are LANai-7's, using a LANai-9 on a 64-bit PCI system improves bandwidth beyond that of point-to-point. We believe that it is due to link level flow control interactions between the LANai-7's and 9's. The effect also occurs when using a LANai-7 to forward between LANai-4's.

4.2 Discussion

Payload caching is relevant to systems with increased I/O capacity for two reasons. The first is that payload caching reduces host-based forwarding latency, by removing a transfer across the I/O interconnect. Second, a payload cache allows more network cards to be placed on the host's I/O interconnect.

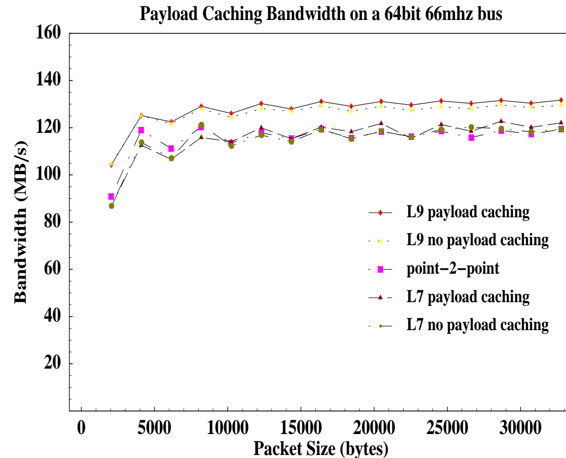

Figure 7. User-level forwarding bandwidth with two versions of Myrinet NIC on a 64-bit 66 Mhz PCI bus.

Though a 32-bit PCI system will not benefit from multiple gigabit NICs with payload caches, a 64-bit system is able to increase forwarding bandwidth beyond 384 MB/s with the host processing 50,000 8KB packets per second and four payload cache enabled LANAi-7 NICs. Two LANAi-9 NICs with link speeds of 250 MB/s are sufficient to push a 64bit 66Mhz system beyond that figure, as long as the host processor does not become the bottleneck or larger packets/MTUs are used. Host-based packet forwarders could forward close to 5 gigabits a second, a quarter of the performance of current high-end custom hardware web forwarding products [6]. Unfortunately, we could not run forwarding tests with all LANAi-9's, as we only had two NICs and their link was clocked at 160MB/s to interoperate with our switches.

5 Conclusion

Packet forwarding, as a local decision within a gigabit LAN, is diverging from packet routing as host-based front-ends are interposed between cluster services and traditional routers to external networks. Such applications include web redirectors, proxies, web servers, or any traffic directing front-end. We present the design, implementation, and user-level performance results of payload caching in Trapeze for three successive generations of Myrinet LANai NIC. The increases in processor clock rate, NIC memory bandwidth, and on-board memory all contribute to the success of payload caching for packet forwarding.

Payload caching is a simple, transparent, and light-weight method of caching incoming packets directly on the

adapter in order to avoid unnecessary host I/O interconnect transfers. It reduces load on the host's I/O interconnect and memory system, alleviating performance bottlenecks in host-based forwarding architectures. Through a simple FIFO based payload cache management scheme, with a small send segment, we observe 100% hit rates forwarding at 106.43MB/s using the next generation Myrinet adapter. Further developments in gigabit networking promise to maintain the host I/O interconnect to network bandwidth ratio close to one, maintaining the relevance of single adapter payload caching. Payload caching enables flexible host-based forwarding agents to compete with mid-class custom forwarding hardware solutions.

References

[1] Darrell Anderson, Jeff Chase, and Amin Vahdat. Interposed request routing for scalable network storage. Technical Report CS-2000-05, Duke University Department of Computer Science, May 2000.

[2] 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.

[3] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing (JSPP 97)*, 1997.

[4] 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.

[5] 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.

[6] Ervin Johnson. A comparative analysis of web switching architectures. At URL http://www.arrowpoint.com/solutions/white_papers/printer/Comparative_Analysis.pdf.

[7] 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.

[8] V. Pai and et. al. Locality-aware request distribution in cluster-based network servers. In *Proceedings of ASPLOS-VIII*, pages 205–216, October 1998.

[9] Larry L. Peterson, Scott C. Karlin, and Kai Li. OS support for general-purpose routers. In *HotOS Workshop*, March 1999.

[10] 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.

[11] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.