Network Interface Data Caching

Hyong-youb Kim, Scott Rixner, and Vijay S. Pai

Abstract—Network interface data caching reduces local interconnect traffic on network servers by caching frequently-requested content on a programmable network interface. The operating system on the host CPU determines which data to store in the cache and for which packets it should use data from the cache. To facilitate data reuse across multiple packets and connections, the cache only stores application-level response content (such as HTTP data), with application-level and networking headers generated by the host CPU. Network interface data caching reduces PCI traffic by 12-61 percent for six Web workloads on a prototype implementation of a uniprocessor Web server. This traffic reduction improves peak throughput for three workloads by 6-36 percent.

Index Terms—Web servers, local interconnects, network interfaces, operating systems.

1 INTRODUCTION

Web server performance has improved substantially in recent years, due mostly to rapid developments in application and operating system software. Techniques such as zero-copy I/O [16], [18] and event-driven server architectures [17] have reduced the CPU load, the amount of main memory used for networking, and the bandwidth requirements of data transfers between the CPU and its main memory. Web servers are now capable of achieving multiple gigabits per second of HTTP content throughput. As server performance increases, the local I/O interconnect within a server, such as the peripheral component interconnect (PCI) bus, has become a potential bottleneck.

Network interface data caching can alleviate the local interconnect bottleneck by caching data directly on a network interface [10]. A software-managed data cache in the network interface stores frequently-served content. The cached content is no longer transferred across the local interconnect. Instead, it is directly transmitted from the network interface, thereby reducing the interconnect traffic. The operating system on the host CPU determines which data to store in the network interface data cache and for which packets it should use data from the cache. Cache contents may be appended to packet-level and application-level headers generated by the host CPU and then sent over the network.

The previously published work on network interface data caching uses a PC-based prototype Web server and shows that caching reduces PCI bus traffic and improves server throughput for several workloads that cause high bus utilization [10]. This paper uses a more recent testbed and further evaluates caching performance on various bus configurations, which include 33 and 66 MHz PCI buses as well as two different system chipsets that provide an interface to the bus and main memory. Small 16 MB caches on network interfaces reduce the server PCI bus traffic by about 12-61 percent for the six workloads used in this paper, regardless of the bus configuration. This reduction in turn improves peak HTTP content throughput by 6-36 percent for the three workloads that cause very high PCI bus utilization. PCI overheads turn out to be a major source of inefficiencies. At least 20 percent of the bus bandwidth is wasted by transfer-related PCI overheads, which are mainly stalls on main memory accesses. Because the bus speed does not affect main memory access latencies, increasing the bus speed also increases data transfer overheads unless the main memory latency improves accordingly. For instance, the 66 MHz PCI bus loses about twice many cycles to the overheads than the 33 MHz PCI bus when the main memory latency remains constant. Although recent interconnects such as PCI-X and PCI Express continue to improve bandwidth through faster clocks, they still incur significant overhead per transfer. Thus, techniques like network interface data caching that enable more efficient use of the available bandwidth may continue to be useful as the host CPU and other system components scale.

The remainder of this paper proceeds as follows: Section 2 describes the flow of request and response data in a Web server. Section 3 explains the concept of network interface data caching and its potential benefits, and Section 4 details the design and use of the cache. Section 5 describes the experimental methodology. Section 6 discusses the experimental results. Section 7 and Section 8 discuss Web workloads and related work. Section 9 draws conclusions.

2 ANATOMY OF A WEB REQUEST

To illustrate the performance issues in Web servers, this section examines the flow of an HTTP request and response through a system that supports zero-copy I/O and includes a network interface card (NIC) that supports checksum offloading. Operating systems that support zero-copy I/O use special APIs or memory management schemes to avoid copying data between the kernel and user space of main

- *H.-y. Kim and S. Rixner are with Rice University, Department of Computer Science, MS 132, 6100 Main St., Houston, TX 77005. E-mail: {hykim, rixner}@rice.edu.*
- *V.S. Pai is with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907. E-mail: vpai@purdue.edu.*

Manuscript received 25 Apr. 2004; revised 17 May 2005; accepted 3 June 2005; published online 16 Sept. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0143-0404.

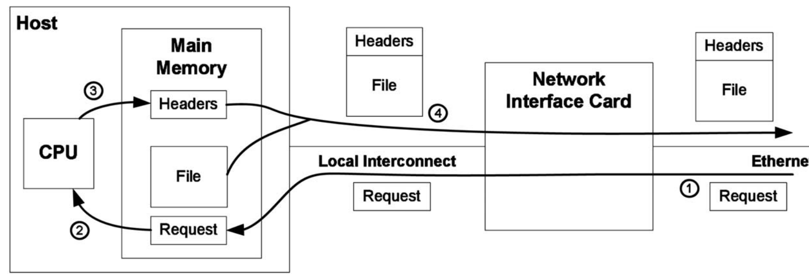


Fig. 1. Steps in processing an HTTP request for a file that currently resides in the operating system file cache.

memory or between different subsystems of the kernel (such as the file cache and network buffers) [7], [16], [18]. With checksum offloading, the NIC, rather than the host CPU, computes various checksums such as TCP/IP checksums [12]. Both zero-copy I/O and checksum offloading reduce the load on the host CPU and are commonly employed in a modern Web server.

Fig. 1 shows the steps taken by a typical Web server to process an HTTP request for a static Web page and to produce a response, assuming that the requested file resides in the file cache. In step 1, the packets that contain the HTTP request arrive on the Ethernet link. The NIC initiates a direct memory access (DMA) across the local interconnect, through which the device takes control of the interconnect and writes the packets into the main memory. In step 2, the operating system running on the host CPU reads the packets from the main memory, validates the packets to ensure that they are not corrupted, and extracts the HTTP request, which is then read by the Web server through a system call. In step 3, the Web server creates appropriate HTTP headers and sends the headers and requested file using a system call. If the file resides in the main memory file cache, then the kernel passes a reference to the file to the TCP/IP network stack (if the file is not found in the main memory, then the file system initiates an I/O operation to read the file from disk). The TCP/IP network stack creates TCP/IP headers and packets that contain HTTP headers and content (the requested file). The device driver then alerts the NIC of new packets to be transmitted. In step 4, the NIC initiates DMA transfers of the TCP/IP headers, HTTP headers, and HTTP content from main memory to the network interface buffers. Finally, the NIC calculates checksums for each packet and sends it out onto the network.

As the server begins to utilize a significant fraction of the Ethernet bandwidth, the local interconnect, such as the popular PCI bus, can become a performance limiter. A standard 64-bit/33 Mhz PCI bus provides a peak bandwidth of 2 Gb/s, theoretically enough to deliver HTTP content through two Gigabit Ethernet NICs at full transmit bandwidth. However, it cannot achieve the peak bandwidth due to the overheads associated with data transfers such as addressing and main memory stalls. These PCI bus overheads in a Web server can consume over 30 percent of the bus bandwidth, so the efficient use of the PCI bus can significantly affect Web server performance.

3 A Network Interface Data Cache

Adding a data cache directly on the NIC allows it to capture repeatedly transferred files. By storing frequently requested files in this network interface data cache, the server will not need to send those files across the interconnect for each request. Rather, the server can simply generate the appropriate protocol and application headers, and the NIC can combine those headers and the file data to be sent out over the network. Referring back to Fig. 1, the Web server normally transfers requested files across the local interconnect to the NIC (step 4 in the figure). Storing copies of files in a cache on the NIC eliminates this final transfer of file data from the system memory to the NIC's local memory, reducing the bandwidth demands on both the local interconnect and main memory.

Fig. 2 shows the stages in processing a Web request in a system with a network interface data cache. Steps 1-3 remain unchanged from those in Fig. 1. In step 4, however, if the operating system determines that the file being sent is currently cached within the NIC, then only the headers (HTTP, TCP, and IP), the location of the cached data in the NIC's local memory, and its length are transferred to the network interface via DMA. In step 5, the NIC then finds the data in its local memory, appends this data to the

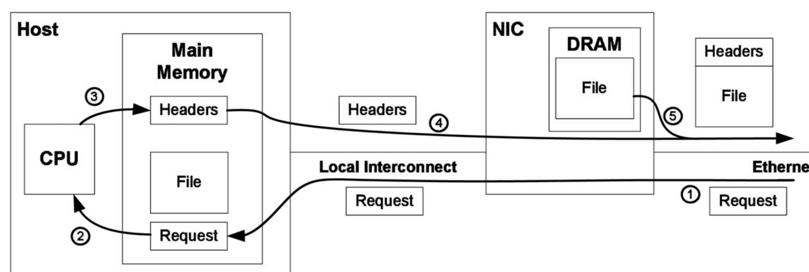


Fig. 2. Steps in processing an HTTP request for a file that currently resides in the network interface data cache.

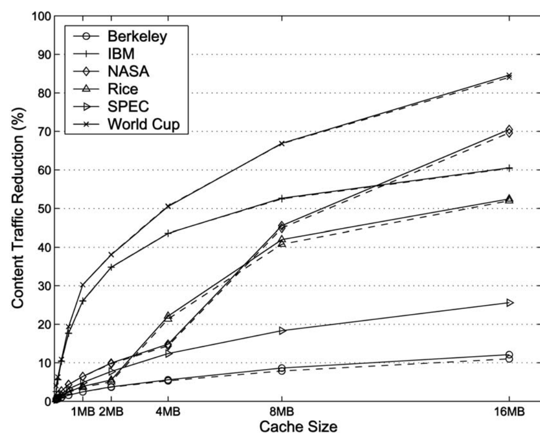


Fig. 3. Potential reduction in HTTP content traffic using LRU caches.

headers to form complete packets, calculates the required checksums, and sends them over the network. A system with a network interface data cache reduces traffic on the local interconnect by transferring only relatively short header information when possible, rather than transferring entire files for every HTTP request.

The network interface data cache may store parts of a file if necessary. When the kernel finds that parts of the requested file reside in the network interface data cache, it informs the NIC of the location of each part in the cache in addition to the location of the remainder that resides in the main memory. The NIC then transfers the remainder from the main memory and assembles complete packets. The other operations of the cache remain the same as described above.

3.1 Content Locality

Due to space, power, and cost constraints, the memory on a NIC is far more limited than the server's main memory, so the network interface data cache will be much smaller than the operating system's file cache. Therefore, for effective caching, Web server requests must have significant data locality. Fig. 3 shows the percentage of the HTTP content traffic that would be eliminated from the local interconnect by network interface data caches of varying sizes. This figure was generated using a cache simulator that simply plays back a Web trace and determines what portion, if any, of each successive requested file is currently in the cache. Files greater than the cache size are not cached. The traces are access logs of Web sites for Berkeley's Computer Science Department, IBM, NASA, Rice's Computer Science Department, and the 1998 Soccer World Cup. The NASA and World Cup traces are publicly available from the Internet Traffic Archive. In addition to these logs from real Web sites, an access log produced by the SPECweb99 benchmark is also used. SPECweb99 evaluates the Web server's capacity by using synthetic clients that generate requests for both static and dynamic content at a fixed rate (the default 400 Kb/s). SPECweb99 was configured to emulate 1,536 clients with the default mix of operations (70 percent static and 30 percent dynamic content). Thus, the SPECweb99 access log contains both static and dynamic content requests, but only the static content requests are fed to the simulator because only static files can be easily cached. The basic statistics of the access logs are shown in Table 1. The figure shows the results for caches sized from 64 KB to 16 MB, with 4 KB blocks using least-recently used (LRU) replacement. The solid lines show the potential traffic reduction for a single cache. The dashed lines show the potential traffic reduction if two caches of the same size are used with the trace split evenly across the two. This simulates the potential caching behavior of a server with two NICs. Even though the use of two caches doubles the total cache size, the traffic reduction is slightly lower since splitting the traces reduces temporal locality. The SPEC trace is an exception, and the lines completely coincide, since the SPEC trace is generated statistically.

The figure shows that dual 16 MB caches can potentially reduce more than 50 percent of HTTP content traffic for those traces that have working sets less than 1 GB (IBM, NASA, Rice, and World Cup). However, for the Berkeley and SPEC traces that have much larger working sets, the caches are less effective, showing about 12 percent (Berkeley) and 25 percent (SPEC) of potential reduction in HTTP content traffic. As mentioned above, the SPEC trace fed to the cache simulator includes only the static content requests. Since they account for 70 percent of an actual SPECweb workload, 25 percent reduction in static content traffic leads to about 17 percent reduction in overall HTTP content traffic. Overall, even just a few megabytes of data cache can significantly reduce HTTP content traffic, indicating substantial potential main memory and local interconnect bandwidth savings.

TABLE 1
Basic Statistics of the Web Server Access Logs Used for the Experiments

Statistic	Berkeley	IBM	NASA	Rice	SPEC	World Cup
Trace Duration	1 month	4 days	1 month	1 month	20 minutes	2 weeks
Year	2000	1998	1995	2000	1999	1998
Total Response Bytes (MB)	82,545	44,487	96,069	8,588	50,644	72,869
Average Response Size (B)	25,920	2,857	56,983	34,936	14,732	6,847
Data Set Size (MB)	6,664	997	271	1,191	5,036	93
Total Requests	3,184,540	15,568,217	1,685,904	245,820	3,604,784	10,641,170
Distinct Files	97,766	39,363	4,690	15,528	33,387	5,163

For the SPECweb99 log, all statistics except the data set size only account for the static content requests.

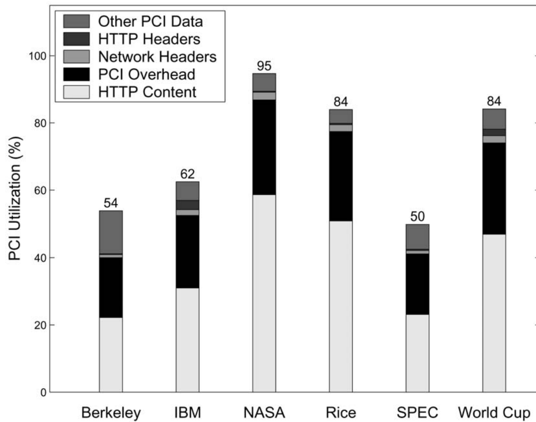


Fig. 4. PCI bus utilization measured in a Web server under various Web workloads without network interface data caches.

3.2 Bus Utilization

The expected reduction in the bus traffic from using network interface data caches depends on the contribution of the HTTP content to the overall traffic as well as the content locality. Fig. 4 shows the utilization of the PCI bus during the execution of each workload on a PC-based Web server without network interface data caches. The server includes a single AMD Athlon XP 2600+ CPU, 2 GB of DRAM, two Gigabit Ethernet NICs on a 64-bit/33 MHz PCI bus, and a PCI bus analyzer (see Section 5 for details). The utilization shown in the figure is the ratio of the measured PCI traffic during the execution of the workload to the peak theoretical traffic for the server's PCI bus during the same time period. The figure categorizes the different sources of PCI utilization on the server, including HTTP response content, PCI overhead, networking headers (TCP/IP and Ethernet headers from the server), HTTP response headers, and other PCI data (including HTTP request headers, TCP/IP packets from the client, and traffic from other peripherals). PCI overhead accounts for bus cycles spent on transfer-related overheads such as address cycles and main memory stall cycles. The NASA trace nearly saturates the bus with 95 percent utilization. However, about 30 percent of bandwidth is wasted due to overheads, so the bus is able to transfer only about 1.2 Gb/s of HTTP content even though its theoretical peak bandwidth is 2 Gb/s. The Rice and World Cup traces also cause high bus utilization—around 84 percent. The IBM trace utilizes only about 62 percent of the bus cycles because the small average response size causes the CPU to quickly saturate. The Berkeley trace requires heavy disk accesses due to its large working set, so disk latency becomes a bottleneck. The SPECweb workload yields the lowest utilization due to dynamic content generation and disk accesses.

Overall, HTTP content and PCI overhead account for about 60 percent and 30 percent of all PCI traffic, respectively, regardless of the workload. Network interface data caching directly targets the HTTP content, the largest component of the PCI traffic. Reductions in HTTP content traffic in turn lead to reductions in PCI overhead, since the system will now handle fewer transfers. Network interface data caching aims to reduce the two components that account for roughly 90 percent of all PCI traffic, so it is expected to achieve large reductions in HTTP content traffic with reasonable storage capacity. Thus, network interface data caching should provide substantial reductions in overall PCI traffic for the workloads studied in this paper.

4 Network Interface Data Cache Design

A network interface data cache utilizes a few megabytes of DRAM added to a NIC with a programmable processor. The cache resides in the DRAM and stores data that may be appended to packet-level and application-level headers generated by the host CPU and then sent out over the network. The operating system running on the host CPU determines which data to store in the network interface cache and for which packets it should use data from the cache.

4.1 Cache Architecture

The network interface data cache is simply a region of local memory (on-board DRAM) on the NIC. Ideally, the cache is as large as possible, but, as shown in Fig. 3, a small 16 MB cache can significantly reduce interconnect traffic. Such a cache can be implemented with a single 16 MB DRAM chip that dissipates less than 1 Watt [13], leading to minimal increases in area and power on the NIC.

Since the cache resides in the local memory of the NIC, only the processor on the NIC may access the cache. However, the network interface data cache is controlled entirely by the operating system on the host processor. The NIC processor acts as a slave to the operating system. It inserts and retrieves information from the cache only at the direction of the host. When adding new data to the network interface data cache, the operating system instructs the NIC to fetch that data from the main memory and store the data at a particular offset in the cache. The NIC then fetches the specified data from the main memory into the cache. When the operating system decides to use data within the network interface data cache for a packet, it simply instructs the NIC to append data from the cache to the packet by giving the offset and length of the desired data in the cache. In this way, the operating system can use any data in the network interface data cache for any outgoing packet. For example, the data can be a subset of a block that was previously inserted into the cache or can straddle multiple cached blocks.

4.2 Cache Management

Since the processor on the NIC does not interpret the data in any way, the host processor must establish policies for allocation, replacement, and use of data in the network interface data cache. Additionally, the host processor must resolve the cache coherence problem that arises on modifications to the main memory copy of content replicated on the NIC local memory. The operating system implements all policies for these cache management tasks.

When allocating storage in the network interface data cache, the operating system caches content at the granularity of a file block. Caching blocks instead of packets allows the TCP/IP stack to structure packet contents differently for different responses, if necessary, and also simplifies cache management by using fixed size objects. The operating

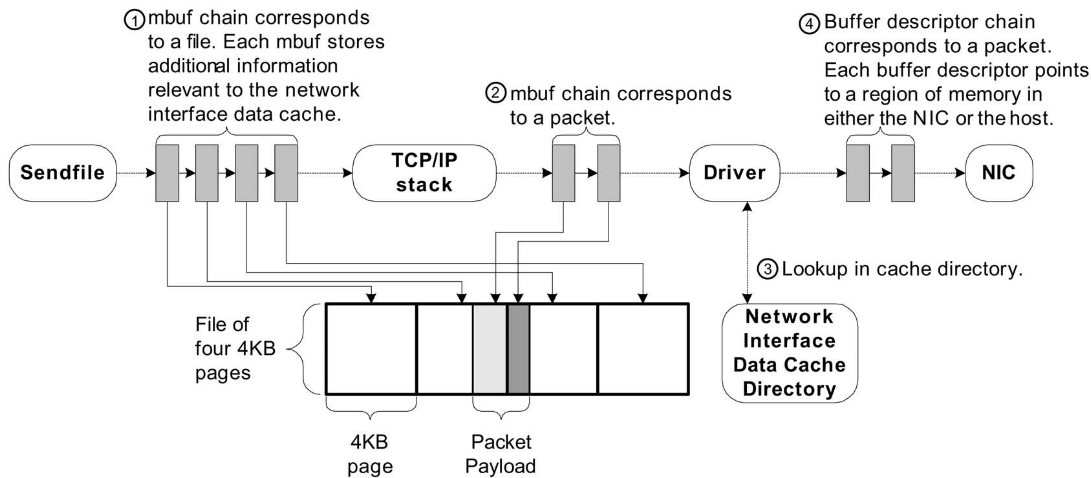


Fig. 5. Steps in sending a response using the `sendfile` system call.

system also manages a directory of the cached blocks. The directory entries contain information relating a cached block to the original file that contains the block. An entry contains the file identifier, the offset within the file, the file revision number (maintained by the operating system to track changes in files), and any required status information associated with the data stored in the network interface data cache. A system with multiple NICs has separate directories for each network interface data cache, since the NICs have separate storage.

The operating system attempts to use data from the network interface data cache in response to the `sendfile` system call from the application. `Sendfile` is a commonly-implemented API for zero-copy I/O in servers. Although a server can also transfer data using the `read` and `write` system calls, the use of user-level data buffers in those system calls commonly causes the kernel to copy data from the kernel file cache to user space on a `read` and from user space to the kernel network buffers on a `write`. Such copying increases CPU and memory load in performing the copies and memory pressure in storing multiple copies of the same information. In contrast, `sendfile` allows for a straightforward implementation of zero-copy I/O since it refers to file content through a descriptor rather than a user-level buffer. The experimental results show that the use of `sendfile` system call improves Web server throughput by up to 47 percent for the Web server workloads used for this study.

Although the system benefits from zero-copy I/O, it is not a requirement for network interface data caching. The network interface data caching can use any API as long as the operating system can relate each piece of data that is being sent out onto the network to the original file and supply the cache directory with information required in directory entries.

Fig. 5 depicts the actions taken by the FreeBSD operating system in response to `sendfile`. If a call to `sendfile` specifies a portion of the file that resides in the operating system file cache, then the operating system creates a set of small memory buffers (called `mbufs` in FreeBSD) to hold control information and a pointer to the data in the file cache. Each `mbuf` specifies a contiguous region of memory. In the figure, each `mbuf` points to a page of the file to be sent. In step 1, the operating system annotates these `mbuf` structures with the original file identifier, the offset into the file (page offset), and the file revision number. The `mbuf` chains created by `sendfile` are transformed into packets by the TCP/IP stack. In step 2, the process of forming packets may split the `mbufs` so that they reference subranges of pages because each `mbuf` can reference at most one contiguous region of memory. The TCP/IP stack then passes the `mbuf` chain for each packet to the device driver for the NIC that will be used for transmission.

In step 3 in the figure, the device driver looks up each referenced block in the network interface data cache directory. If the block is already present, then the driver informs the NIC of the offset and length of the content within the network interface data cache. If the block is not present, the driver allocates a block in the cache, using a replacement policy to evict old blocks if no space is available. In either case, the driver creates a set of buffer descriptors to pass the relevant information to the NIC (step 4). Each buffer descriptor either points to a main memory buffer or a region of the network interface data cache. The CPU typically notifies the NIC that it has created new buffer descriptors by writing to a memory mapped register of the NIC. The NIC then retrieves the buffer descriptors using DMA and uses the information contained within them to initiate the necessary DMA transfers to retrieve the data from main memory. After completing the requested operation, the NIC interrupts the CPU to inform it that the buffer descriptors have been consumed. In a system without a network interface data cache, these buffer descriptors always require the NIC to transfer packet data from main memory using DMA. In a system with a network interface data cache, however, buffers pointing to cached content require no DMA transfers.

The file revision field stored in each directory entry enables a straightforward mechanism to keep the cached blocks coherent with the objects stored on the server's main storage system. When looking up blocks in the network interface data cache directory, the device driver lazily

- `nic_cache_init()`
  Allocate and initialize the data cache on the programmable network interface and return its size.
- `nic_cache_insert(mmaddr, ncoffset, len)`
  The buffer descriptor contains `mmaddr`, `ncoffset`, and `len`. Use DMA to transfer `len` bytes starting at main memory address `mmaddr` into the network interface data cache starting at offset `ncoffset`.
- `nic_pkt_append_mm(mmaddr, len)`
  The buffer descriptor contains `mmaddr`, `len`, and additional flags. Use DMA to transfer the `len` bytes starting at main memory address `mmaddr` into the network interface's transmit buffer. This function is unchanged from the original operation of the NIC.
- `nic_pkt_append_cache(ncoffset, len)`
  The buffer descriptor contains `ncoffset`, `len`, and additional flags. Copy `len` bytes from the cache starting at offset `ncoffset` into the transmit buffer.

Fig. 6. Commands supported by the programmable processor on the network interface and invoked by the operating system.

invalidates blocks for which the current revision identifier does not match the cached revision. Note that multiple NICs do not present any additional cache coherence problems, since each network interface data cache operates independently.

To further simplify the coherence implementation, the operating system keeps the network interface data cache strictly a subset of the main memory file cache. With this guarantee of inclusion, information mapping file blocks to network interface data cache storage need not persist beyond the replacement of a block from the main memory file cache. This inclusion property provides two further benefits. First, it allows caching even for NICs that do not support checksum offloading, since all content on the NIC also resides in the main memory and can thus have checksum calculations performed by the host CPU. Second, inclusion simplifies network retransmits in the event of replacements from the network interface data cache, since the the operating system always keeps data in the main memory file cache for retransmits.

The cache management described so far ignores possible synchronization issues that may arise in a multiprocessor system. The design requires little additional synchronization to an operating system that already supports multiple processors for the following reasons. First, the design makes use of existing data structures of the operating system, except the cache directory in the device driver. Second, the execution flows of the network stack with and without network interface data caches are essentially the same. Assuming that the original network stack is properly synchronized to support multiple processors, only accesses to the cache directory require additional synchronization. If accesses to the device driver are already synchronized, even this synchronization is unnecessary.

4.3 Cache Interface

The operating system manages the network interface data cache using an API that consists of the four functions listed in Fig. 6. The figure only shows the functions used to send packets; the NIC must also support functions to receive packets and perform other actions, but those remain unchanged because the cache does not affect those tasks. Since the host processor cannot directly call functions on the NIC's processor, these API functions are actually implemented using existing mechanisms to communicate from the host processor to the NIC. In particular, the operating system uses flags in the buffer descriptor data structure to indicate which command to invoke, and additional fields within those buffer descriptors to pass arguments to the NIC.

The API for the network interface data cache includes functions to initialize the cache, to copy data from main memory to the network interface data cache, to append a block of main memory to the current packet, and to append a cached block to the current packet. All other NIC functions remain unchanged. The initialization function, `nic_cache_init`, allocates space in the NIC's local memory and notifies the operating system of the amount of memory that has been allocated so that the operating system may construct and manage the cache directory. Data is added to the cache using the `nic_cache_insert` function which transfers the data from main memory through DMA. As with all DMA transfers, a single buffer descriptor can only describe one contiguous buffer. If disjoint memory regions are to be added to the cache, the operating system must call `nic_cache_insert` multiple times. The API does not include an explicit invalidation command. Instead, the operating system simply invalidates its directory entries.

The API of a conventional NIC effectively only includes the `nic_pkt_append_mm` function to construct and send packets. As described in Section 4.2, the operating system transmits packets by generating a list of buffer descriptors that are then fetched by the NIC. Each buffer descriptor points to a main memory buffer that the NIC should append to the current packet in the NIC's transmit buffer by using a DMA transfer. Such DMA transfers are initiated by invoking the `nic_pkt_append_mm` function on the NIC for each buffer descriptor. Additional flags in the buffer descriptor are used to indicate to the NIC if that block is the first or last block in the packet, if a particular function should be performed before sending the packet (such as checksum offloading or other future services), or any additional information required by the NIC to process the packet.

TABLE 2
Server Configurations Used to Evaluate Network Interface Data Caching

Configuration	Host Processor (Clock Speed)	PCI Bus (Maximum Throughput)
AMD/33	AMD Athlon XP 2600+ (2.1 GHz)	64-bit/33 MHz PCI Bus (2.1 Gb/s)
AMD/66	AMD Athlon XP 2600+	64-bit/66 MHz PCI Bus (4.2 Gb/s)
Intel/33	Intel Xeon (2.4 GHz)	64-bit/33 MHz PCI Bus (2.1 Gb/s)
Intel/66	Intel Xeon	64-bit/66 MHz PCI Bus (4.2 Gb/s)

The remainder of the text uses short configuration names to concisely describe server hardware.

The last API function `nic_pkt_append_cache` resembles `nic_pkt_append_mm` but copies data to the transmit buffer from the indicated offset in the network interface data cache, not from main memory through DMA. This copy on the network interface could also be eliminated if the medium access control (MAC) engine that transfers the data out onto the network could gather the packet from disjoint memory regions on the NIC. As with `nic_pkt_append_mm`, additional flags in the buffer descriptors are used to indicate if the block is the last block in the packet or if additional processing should occur.

The NIC executes commands in the same order as they are issued by the driver. The NIC also interlocks the commands appropriately in order to resolve any data dependencies that may arise when multiple commands are pending. For instance, when executing `nic_pkt_append_cache` followed by `nic_cache_insert` to the same cache block, the NIC delays the transfer of new data (the second command) until the copying of the block completes (the first command).

Referring back to Fig. 5 in Section 4.2, the buffer descriptors of step 4 convey the commands of Fig. 6. If the data represented by an `mbuf` is not cached, the driver inserts the data into the cache using `nic_cache_insert`. The driver then sends a series of buffer descriptors that contain command `nic_pkt_append_cache` or `nic_pkt_append_mm`, as appropriate, to transmit each packet. The processor on the NIC concatenates the cached data with the headers fetched from the main memory before the packets are transmitted onto the network. Headers for TCP/IP, Ethernet, and HTTP are transferred to the NIC using `nic_pkt_append_mm` and are never inserted into the network interface data cache since they do not refer to a file block. Similarly, other types of data such as dynamically generated HTTP content that do not refer to a file block are also transmitted using only `nic_pkt_append_mm`.

5 Evaluation Methodology

5.1 Prototype Implementation

A prototype implementation of network interface data caching is built using a PC-based server and programmable Gigabit Ethernet NICs. The prototype implements an LRU block cache with lazy invalidation, a block size equal to the page size of the operating system (4 KB), and a requirement of inclusion in the main memory file cache. It does not cache files greater than the cache size. The LRU replacement policy and 4 KB block size are chosen to simplify the implementation. The server employs four different configurations of the host processor and PCI bus in order to evaluate the impact of technology on performance of network interface data caching. Table 2 shows the configurations. The AMD servers use the Tyan Tiger MPX motherboard based on the AMD-760 MPX chipset, and the Intel servers use the Tyan Tiger i7500 motherboard based on the Intel E7500 chipset. All server configurations have a single processor, two 1 GB PC2100 DDR SDRAM DIMMs, two 36 GB SCSI disks, a SCSI disk controller plugged into a 32 bit PCI slot, two 3Com 710024 copper Gigabit Ethernet NICs plugged into 64 bit PCI slots, and a VMETRO PBT-615B PCI bus analyzer. The Intel Xeon processor supports Hyper-Threading [9], but this feature has been disabled so all of the results consistently reflect uniprocessor performance.

The PCI bus analyzer passively measures the PCI bus utilization and injects no traffic onto the PCI bus. Both the AMD and Intel systems have multiple PCI buses, but they are organized differently. On an AMD system, the single analyzer can monitor the NICs and the SCSI controller. However, on an Intel system, the NICs and the SCSI controller are on two independent bus segments such that the single analyzer captures all NIC traffic but none of the SCSI traffic. Because network interface data caching only affects HTTP content traffic transferred to the NICs and its associated PCI overhead, it has no impact on the SCSI traffic. Likewise, the SCSI traffic has no impact on caching performance. Thus, capturing the NIC traffic is sufficient for evaluating caching effects on bus traffic.

The 3Com 710024 NIC is based on the programmable Tigon Gigabit Ethernet controller and has 1 MB of on-board memory. The controller includes two simple MIPS-based programmable cores. The 3Com NIC runs a modified version of Revision 12.4.13 of the Tigon firmware, which was made open-source by the manufacturer [1]. The modified firmware implements the API commands of Section 4.3 and various optimizations to reduce the possibility of the NIC being a potential bottleneck. Specifically, the modified firmware parallelizes tasks across the two processors and sets tunable parameters to communicate with the host more efficiently [11]. Note that the cache content is only accessed by the DMA and MAC hardware, so task parallelization does not introduce synchronization problems for the processors. The processors do need to maintain small data structures in order to resolve data dependencies among multiple pending commands that use or modify the cache, as mentioned in Section 4.3. In the current firmware implementation, this task is handled entirely by one processor. Although the current prototype uses a MIPS-based programmable controller, network interface data caching may be implemented

on any other programmable controller. The specific NIC used in this study has been discontinued, but Broadcom continues to produce Gigabit Ethernet controllers that are very similar to the Tigon. Unfortunately, the specifications of these newer controllers are not publicly available.

The server runs a version of the FreeBSD 4.7 operating system with the following modifications. The `sendfile` system call is extended to use network interface data caching. The `mbuf` structure has five new fields, and `mbuf` manipulation routines are modified to handle the new fields appropriately. Finally, the device driver for the NIC is modified to maintain cache directories and generate modified buffer descriptors that contain the commands for network interface data caching.

The 3Com NIC only has 1 MB of on-board memory. At minimum, roughly three-fourths of the on-board memory are required for the firmware code, transmit buffer, and receive buffer. Thus, the 1 MB of on-board memory is insufficient to evaluate network interface data caching. Instead, network interface data caches of various sizes are emulated using the following modifications to the API shown in Fig. 6. Upon receiving a `nic_cache_insert` command, the prototype fetches the specified data and discards it instead of adding it to the cache. On a `nic_pkt_append_cache` command, the prototype simply increments the pointer to the end of the transmit buffer by the specified length, using whatever data is currently in the buffer. The other API functions behave as specified. The NIC with these simplifications generates the same amount of PCI bus traffic and Ethernet traffic as a fully functional NIC that actually stores cached blocks and reuses them on appropriate commands. However, the lack of copying in `nic_pkt_append_cache` ignores the overhead of copying. This copying is unnecessary for a NIC that supports gather I/O and can transmit packets consisting of noncontiguous memory regions. Another problem with `nic_pkt_append_cache` is that packets that include cached data have invalid checksums. The Tigon checksumming hardware is integrated into the DMA engine (this is not to be confused with a manufacturer-documented hardware checksum bug that arises when both read and write DMA transfers are active). Therefore, only data that is transferred between the host and the Tigon may be checksummed. A slight modification to the Tigon architecture to allow data stored in local memory to be run through the checksumming hardware can solve the checksum problem. These additional features are simple and would not require substantial implementation costs. Thus, despite the simplifications in `nic_cache_insert` and `nic_pkt_append_cache` commands, the prototype should accurately emulate network interface data caching and its impact on server performance.

5.2 Test Platform

The performance testbed consists of the prototype Web server and two client machines. Each client machine has two Gigabit Ethernet NICs and runs FreeBSD 4.7. The machines are connected through two Netgear GS508T Gigabit Ethernet switches and use two subnets. Each machine has one NIC on each subnet. The NIC drivers on the clients are modified to accept packets from the server containing cached data despite their invalid checksums discussed in Section 5.1; to support this distinction, the server marks such packets with an artificial time-to-live field in the IP header.

The server runs a version of the Flash Web server that uses the `sendfile` API and supports HTTP pipelined persistent connections [17]. Server throughput is measured using the Web workloads listed in Table 1. For all workloads except SPEC, a trace replayer tool is used. It reads a Web trace and simulates multiple users by opening multiple simultaneous connections to the server, each of which corresponds to a single user. The replayer uses an infinite-demand model, issuing requests as fast as the server can sustain. Requests that came from the same anonymized client IP address within a fifteen second period in the original access log are treated as a single persistent connection. Within a persistent connection, requests that arrive less than five seconds apart in the original log are grouped, and requests within each group are pipelined. Each client machine runs two instances of the replayer, one on each subnet. Requests are split equally among all replayers.

SPECweb99 also simulates multiple clients but does not follow an infinite-demand model. Rather, each client tries to maintain a fixed 400 Kb/s. The server capacity is then measured as the number of clients that achieve at least 320 Kb/s. The results presented in this paper are based on runs in which all clients achieve at least 320 Kb/s and use the default mix of requests (70 percent static and 30 percent dynamic content). The AMD and Intel systems use 1,536 and 1,664 clients, respectively.

For some workloads, the results presented in the following sections are markedly different from the previously reported results from a similar testbed [10]. This study uses a faster CPU, more efficient Web server software (multithreaded Flash versus single-threaded thttpd), and an improved version of `sendfile`. In general, these improve throughput. The throughput for the Berkeley trace improves significantly because multithreaded Flash can overlap disk accesses with network transfers, whereas single-threaded thttpd blocks on disk accesses.

6 Experimental Results

6.1 Local Interconnect Traffic and Server Throughput

Fig. 7a shows the impact of network interface data caching on PCI traffic for AMD/33. The figure shows four bars for each workload. The leftmost bar represents traffic without caching, and the right three bars represent traffic with cache sizes of 4 MB, 8 MB, and 16 MB per network interface. All measures of PCI traffic are normalized to the traffic without caching. As in Fig. 4, each bar is split into five categories. Fig. 7a shows that network interface data caching reduces the HTTP content traffic on the PCI bus, by substantial margins for all workloads except Berkeley and SPEC, as predicted by Fig. 3. Removing HTTP content transfers also reduces the PCI overhead associated with those transfers since the overhead stems from data transfers across the bus. As the HTTP content traffic reduction increases, the PCI overhead decreases, leading to further reductions in PCI traffic.

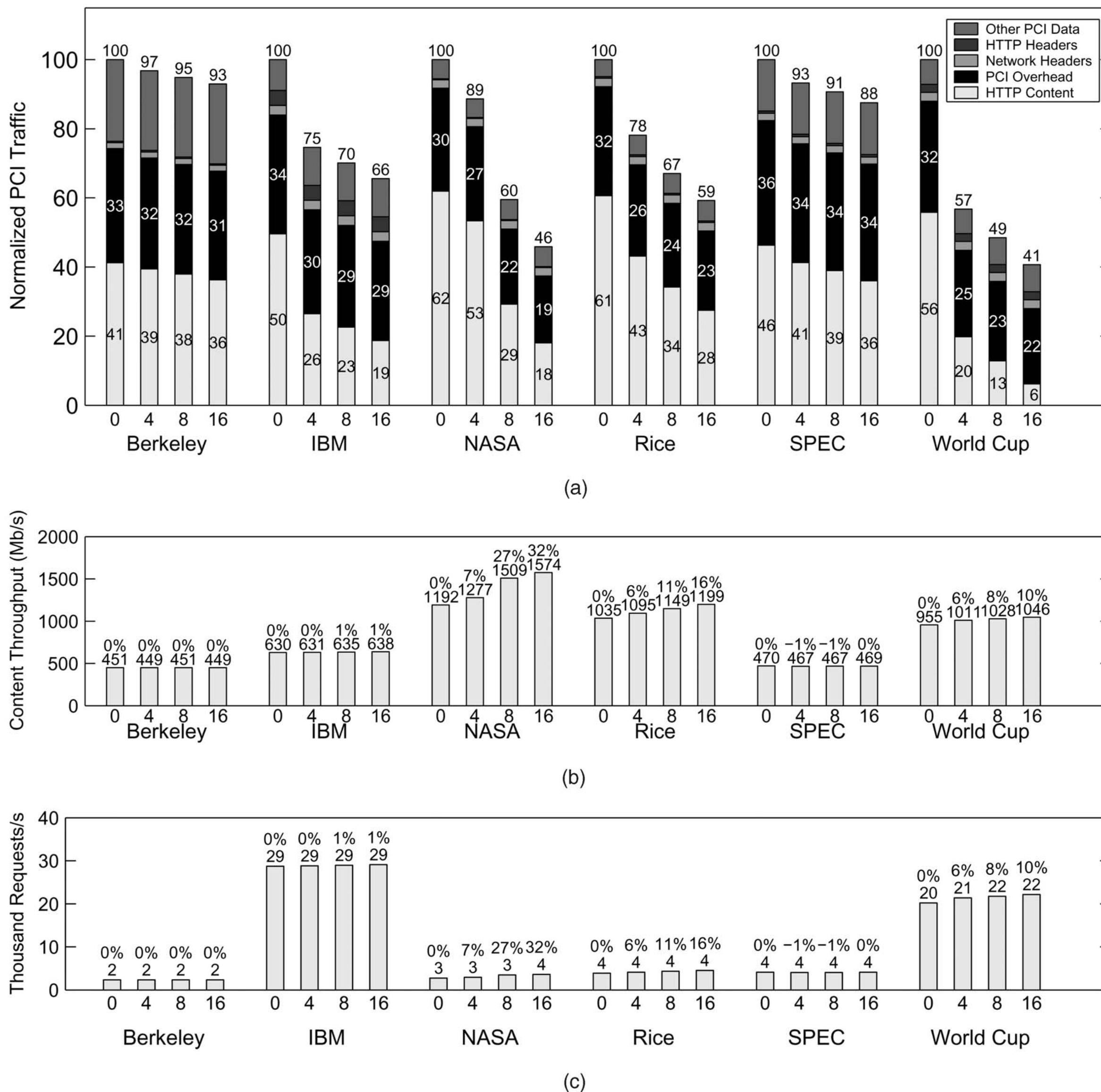


Fig. 7. (a) Bus traffic reduction, (b) HTTP content throughput improvement, and (c) request rate improvement from network interface data caching for AMD/33.

With 16 MB data caches, the server reduces HTTP content traffic on the bus by 55-89 percent for the IBM, NASA, Rice, and World Cup traces. These reductions closely match the predictions in Fig. 3, even though the real system may reorder requests and responses due to various latencies in the system and the different ways of splitting the traces. The PCI overhead accordingly decreases by 16-35 percent, leading to combined overall PCI bus traffic reductions of 34-59 percent. Since only HTTP content is cached on the NIC, network interface data caching does not change the other types of PCI traffic such as HTTP and network headers.

The Berkeley and SPEC workload show little reductions in overall PCI traffic because their large data set sizes (over 5 GB, as shown in Table 1) allow network interface data caching to eliminate only about 10 percent of the overall PCI traffic. More intelligent replacement policies may provide additional benefits for such workloads [4], [6]. Additionally, predicting reuse patterns could allow for reducing cache pollution by bypassing the cache entirely for some data, as has been studied in other contexts [22].

Fig. 7b shows the server throughput improvements (shown in percent) that result from the reduction in PCI traffic discussed above. As shown in Fig. 4, the NASA workload nearly saturates the PCI bus with 95 percent bus utilization without caching. Therefore, caching yields the most benefit, about 32 percent throughput improvement using 16 MB caches. This enables the server to achieve a peak throughput of 1,574 Mb/s on the NASA trace. The Rice and World Cup workloads show the second highest bus utilization (84 percent) without caching. Accordingly, they benefit less from caching than the NASA workload. Throughput improvements are about 16 percent for the Rice trace and 10 percent for the World Cup trace using 16 MB caches. Network interface data caching is most effective at capturing the locality of the World Cup trace, with 16 MB caches reducing 89 percent of the HTTP content traffic. However, throughput improvement for the World Cup

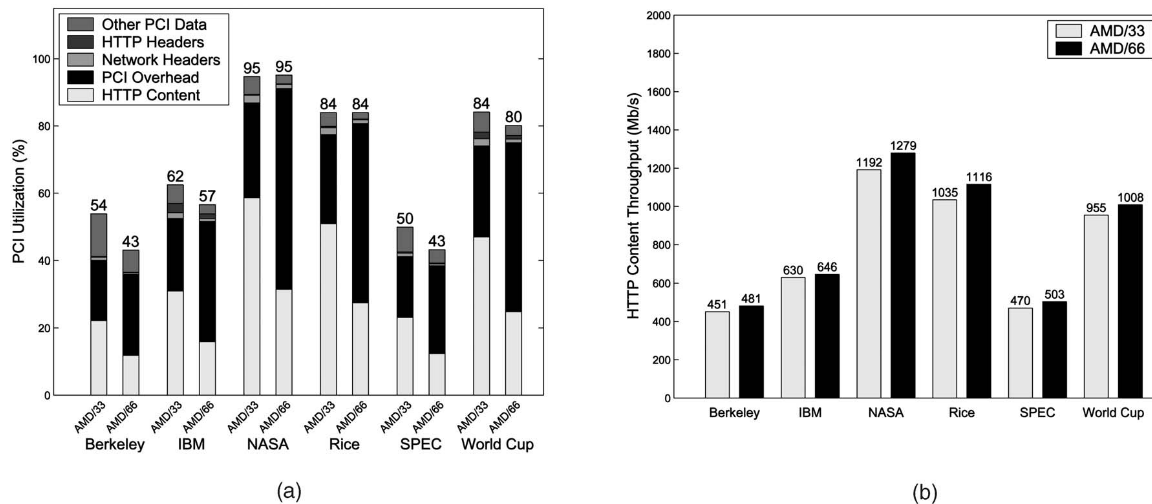


Fig. 8. (a) Comparison of PCI bus utilization and (b) HTTP content throughput achieved with AMD/33 and AMD/66.

trace is less than that for the Rice trace, even though the World Cup trace shows a greater reduction in HTTP content traffic and both achieve the same bus utilization without caching. The main difference between the two is that the Rice trace has larger response sizes. This result indicates that the response size plays a greater role in determining throughput improvements from caching than the traffic reduction. As expected, improvements in request rates from caching, shown in Fig. 7c, are identical to throughput improvements because the request rate should be linearly proportional to the throughput under the infinite demand model. Network interface data caching does not improve the server throughput for the Berkeley, IBM, and SPEC workloads. This is expected since the PCI bus is not a bottleneck in the system without caching. The overhead of managing the cache and using the cache commands slightly degrades server throughput.

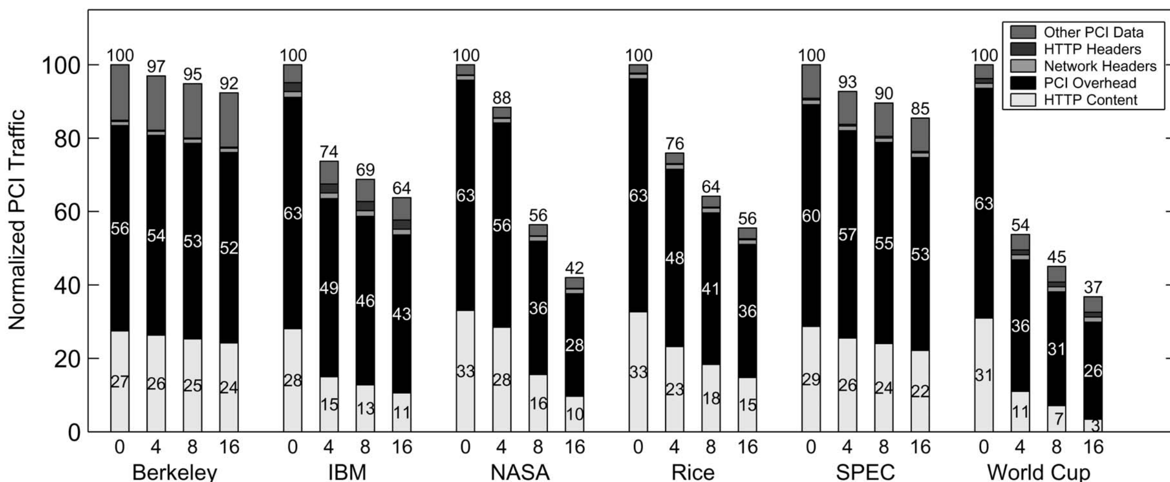
These reductions in PCI traffic and throughput improvements have several consequences. First, systems that are limited by the achievable PCI bandwidth will improve their performance commensurate with the reduction in bus traffic. Second, systems that are not limited by the achievable PCI bandwidth will be able to scale other resources in the system beyond the limits currently imposed by the local interconnect. Scaling other resources such as memory and CPU would increase contention for the PCI bus, and then the system can employ network interface data caching to further improve performance. In both cases, the potential to extract greater performance from existing shared I/O interconnects makes more radical changes to local I/O interconnect designs less attractive because of the additional engineering costs they impose in redesigning motherboards, peripheral interfaces, interconnection components, and operating systems.

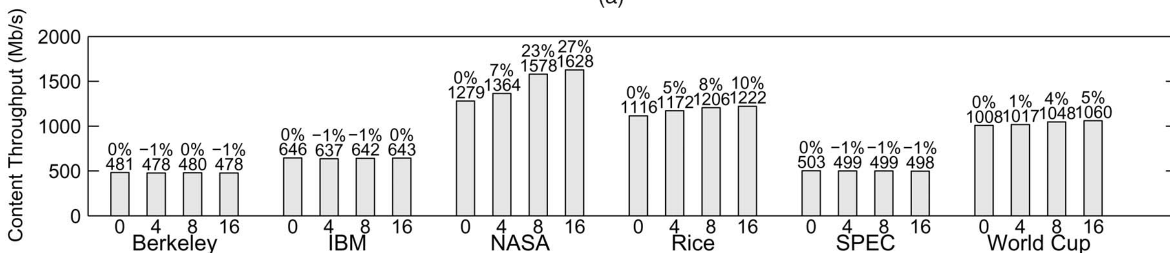
6.2 Interconnect Bandwidth

Network interface data caching relieves the interconnect bottleneck by reducing the required bandwidth. A faster bus can also relieve this bottleneck by increasing the available bandwidth. This section examines the impacts of bus speeds on bus utilization and caching performance by comparing AMD/33 against AMD/66.

Fig. 8a compares the PCI bus utilization of AMD/33 and AMD/66. Because the 66 MHz PCI bus theoretically doubles the bandwidth of the 33 MHz PCI bus, AMD/66 is expected to show lower bus utilization than AMD/33 given the same server throughput. However, both AMD/33 and AMD/66 show almost same bus utilization for the NASA, Rice, and World Cup workloads, while AMD/66 achieves only marginally higher throughputs than AMD/33, as shown in Fig. 8b. These workloads again generate significant bus traffic with overall utilization reaching 80 percent and above. Despite high bus utilization, almost 60 percent of all PCI traffic is consumed by overheads, and HTTP content accounts for only about 30 percent. Remember that on AMD/33 the PCI overhead and HTTP content account for about 30 percent and 60 percent of all PCI traffic, respectively. So, per-byte PCI overhead (PCI overhead divided by HTTP content) is about four times greater on the 66 MHz PCI bus than on the 33 MHz PCI bus. Consequently, even for the most bus-limited workload (NASA), the server throughput increases only by 87 Mb/s as the PCI bus speed increases from 33 MHz to 66 MHz.

Since AMD/33 and AMD/66 achieve similar bus utilization and server throughput without caching, caching is expected to yield similar benefits for both systems. Reductions in the HTTP content traffic shown in Fig. 9a are same as those for AMD/33 shown in Fig. 7a because both servers transfer the same amount of HTTP content, and PCI bus speeds should not affect the potential reduction in HTTP content from caching. The overall PCI traffic reductions are also roughly same as those for AMD/33 shown in Fig. 7 due to the increased PCI overhead associated with data transfers. Figs. 9b and 9c show that the throughput and request rate improvements from caching for AMD/66 are also similar to those for AMD/33. Overall, improvements are only about 5 percent lower than those for AMD/33.





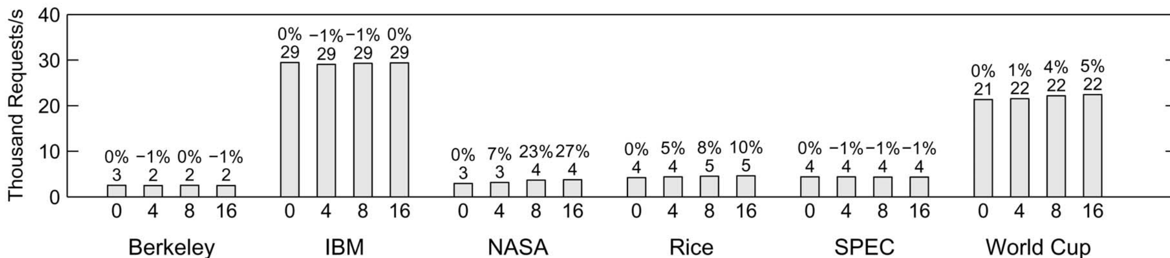


Fig. 9. (a) Bus traffic reduction, (b) HTTP content throughput improvement, and (c) request rate improvement from network interface data caching for AMD/66.

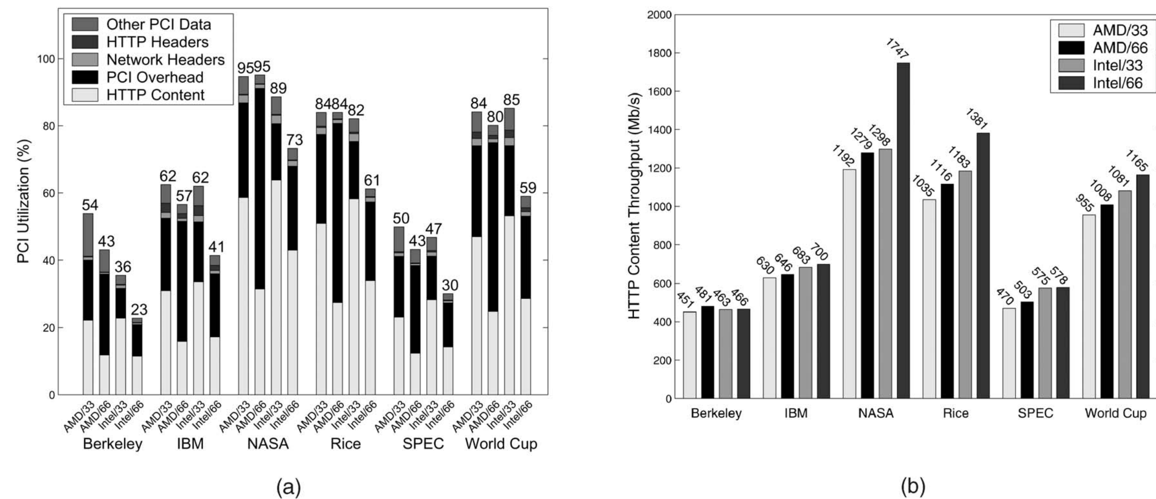


Fig. 10. (a) Comparison of PCI bus utilization and (b) HTTP content throughput achieved with AMD/33, AMD/66, Intel/33, and Intel/66.

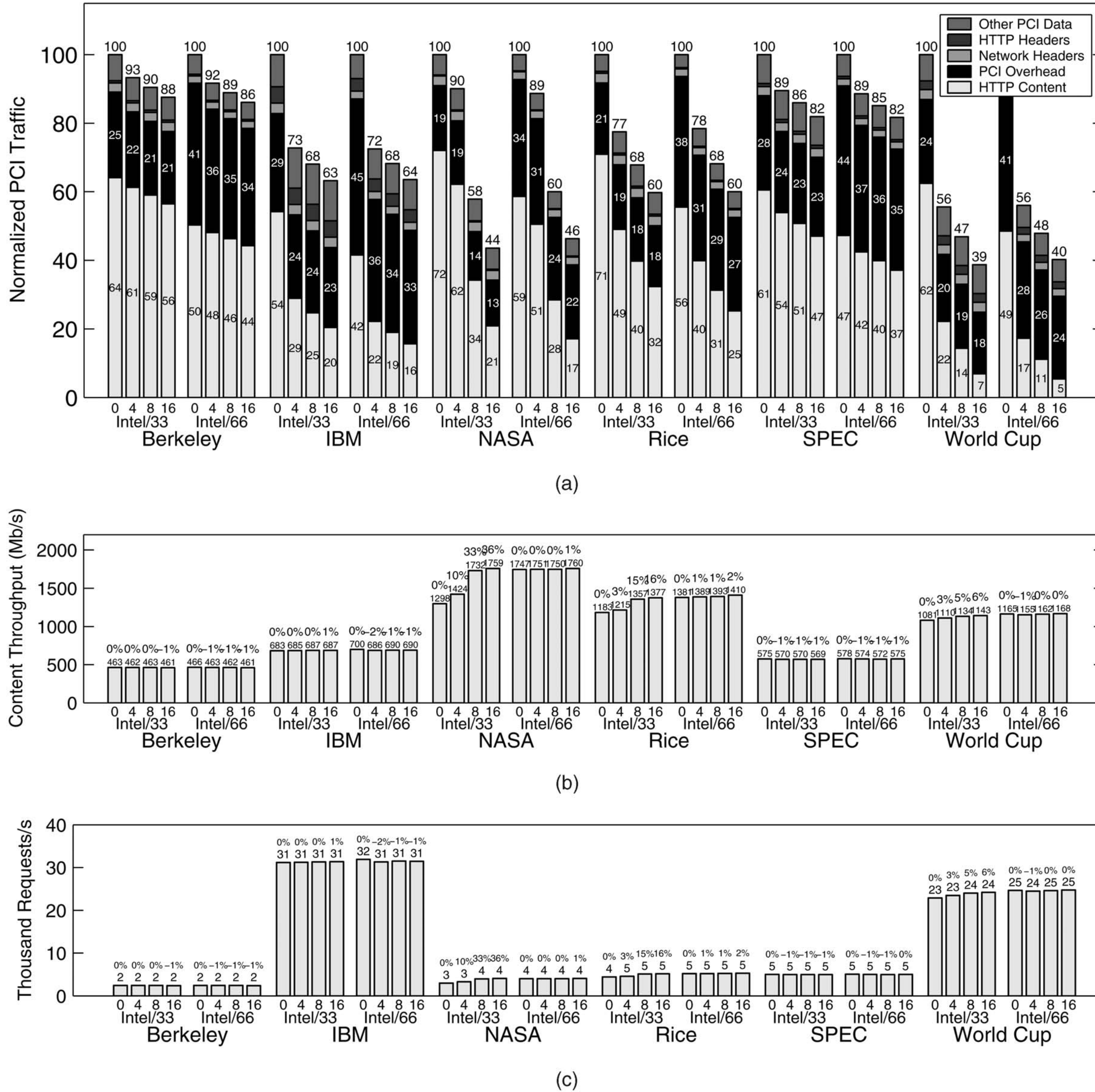


Fig. 11. (a) Bus traffic reduction, (b) HTTP content throughput improvement, and (c) request rate improvement from network interface data caching for Intel/33 and Intel/66.

6.3 Interconnect Interface

Comparison between AMD/33 and AMD/66 shows that per-byte PCI overhead increases nearly four times as the bus speed doubles. The analyzer reveals that the overhead cycles are mostly main memory stalls. Since memory accesses are performed by the system chipset not by the PCI bus, bus speeds should not affect memory access time. Thus, one can expect that doubling the bus speed would also double PCI overhead. A fourfold increase indicates possible chipset performance issues.

Fig. 10 compares the PCI bus utilization and HTTP content throughput of the Intel systems as well as the AMD systems. As explained in Section 5.1, the PCI traffic of the Intel systems in the figure does not include the disk traffic generated by the SCSI controller. This absence of the disk traffic explains the decreases in other PCI data for the Berkeley and SPEC workloads on the Intel systems. Note that only these workloads generate noticeable disk traffic. Overall, AMD/33 and Intel/33 show similar bus utilization and server throughput. Intel/33 generates slightly more HTTP content traffic and higher server throughput than AMD/33, which is expected from a faster CPU. It also shows smaller per-byte PCI overhead. However, Intel/66 shows very different bus utilization from AMD/66. Intel/66 shows much smaller per-byte PCI overhead than AMD/66, so the overall bus utilization is much lower as well. Unlike the AMD systems, per-byte PCI overhead on Intel/66 is only about twice the overhead on Intel/33, which is expected from doubling the bus speed. This allows the most bus-limited workload (NASA) to nearly saturate the two NICs on Intel/66 with 1747 Mb/s of HTTP content throughput.

Bus traffic reductions and consequent server throughput improvements from caching shown in Fig. 11 are as expected. Both Intel/33 and Intel/66 show roughly same bus traffic reductions, as do AMD/33 and AMD/66.

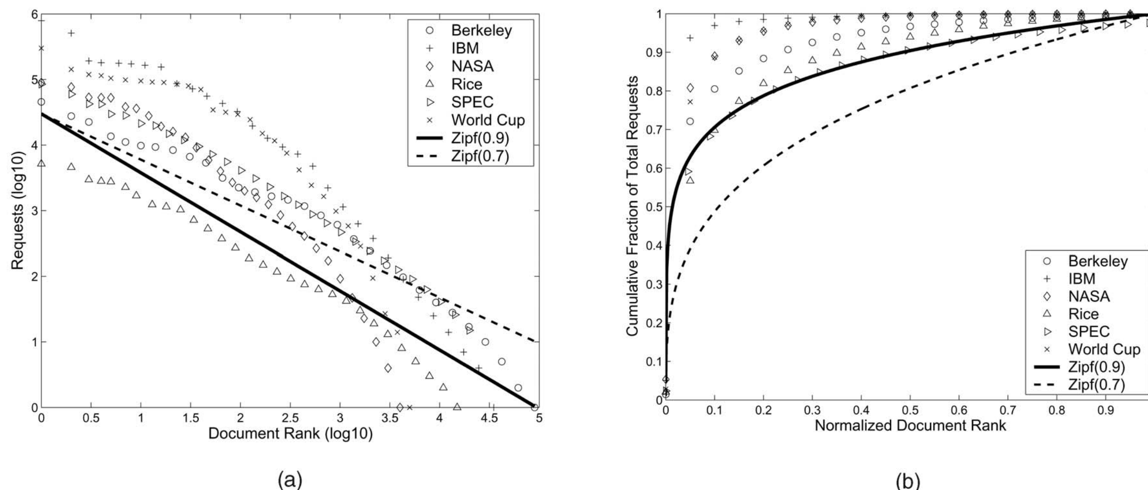


Fig. 12. Comparison of requests and document ranks.

Throughput improvements from caching on Intel/33 are similar to AMD/33 since they show similar bus utilization and server throughput without caching. Intel/66 gains no throughput improvements from caching because the bus is not a bottleneck in the system.

While the Intel systems have lower bus overhead than the AMD counterparts, overhead due to main memory stalls still consumes a significant fraction of bus cycles. Increasing the bus speed provides more available bandwidth. However, because the bus speed does not affect main memory access time, doing so also increases overhead unless memory latencies improve accordingly. Thus, increasing the bus speed alone may not be able to eliminate the potential interconnect bottleneck.

7 Web Workloads

The Web workloads used so far are several years old and may not represent more recent workloads. There are three main concerns. First, all workloads except SPECweb99 only consist of requests for static content. Web sites serve more dynamic content now than several years ago, but static content such as images, scripts, and text is still a significant part of overall HTTP content. Some Web servers are now dedicated to serving static content in order to facilitate dynamic content generation on different servers. Such servers can directly benefit from network interface data caching.

Second, Web requests are believed to follow a Zipf-like distribution in which popular objects receive a large number of requests, thereby enabling effective caching. Fig. 12a shows the number of requests each document (HTTP object) receives. The document rank is a natural number in which 1 represents the most popular object. The Zipf lines are based on request frequencies computed using $(1/R)^{\alpha}$ where R is the rank and either $\alpha = 0.9$ or $\alpha = 0.7$ shown in parenthesis. SPECweb99 uses 1,536 clients as in Section 3.1. The Rice and SPEC curves descend roughly at the same rate as the Zipf(0.9). However, all the other curves show a steeper descent, indicating that popular objects of these workloads receive a greater fraction of requests than Rice or SPEC. The cumulative fraction of total requests shown in Fig. 12b confirms the case. While 16 MB caches can remove substantial HTTP content traffic for the workloads used in this paper, caching may be less effective for workloads whose Zipf-like behavior is less prominent (see Zipf(0.7)).

Finally, the working set size may be correlated to the server throughput. For instance, SPECweb99 increases the data set size as the request rate increases. The other workloads have fixed size data sets. It is unclear whether one must grow in proportion to the other. It is intuitive to believe that more clients with faster connections would demand more diverse objects. It is also intuitive that clients may only be interested in the same set of objects regardless of the number of clients or connection speeds. For instance, the World Cup Web site supports the latter.

Overall, not all the workloads may be realistic, but they tend to follow commonly known characteristics of Web workloads—they can be effectively cached and have Zipf-like distributions. This paper shows that some workloads can cause high bus utilization, and for such workloads, network interface data caching can alleviate the bus bottleneck and improve server throughput.

8 Related Work

Yocum and Chase proposed payload caching to improve packet forwarding performance of network intermediaries such as firewalls and routers by caching packet payloads instead of file blocks [25]. Incoming packet payloads are stored in a payload cache on the NIC so that they may be transmitted directly from the NIC if they are forwarded later from the same NIC. Also, if a packet's payload is not found in the cache when it is sent, it can be added into the cache for later retransmission.

Programmable network interfaces have been used extensively for cluster computing applications. They often implement message passing protocols and zero-copy I/O APIs in order to minimize communication latency. Myrinet [3] and Quadrics [19] use programmable interfaces to implement parts of their message processing and provide

the application with zero-copy I/O APIs. Arsenic [20], Ethernet Message Passing [21], Queue Pair IP [5], U-Net [23], and Virtual Interface Architecture [8] also make use of programmable interfaces to facilitate protocol processing and to support zero-copy I/O. Just as programmable network interfaces help improve cluster computing applications through their extended services, network interface data caching exploits their programmability to improve network server performance.

Coherent network interfaces take advantage of cache coherent interconnects in order to reduce overheads involved in transferring messages in multiprocessor systems [15]. Others have suggested attaching the network interface directly to the memory module, essentially on the coherent interconnect, to overcome the interconnect bandwidth and latency [14]. With a coherent interconnect, the NIC can perform caching operations without any modifications to the operating system. As the NIC fetches data from main memory, it can decide whether to cache it for future uses. The NIC then should be able to determine the staleness of the cached data through the coherence protocol. It would be costly for an I/O interconnect and peripheral devices to implement complex coherence protocols. However, as network speeds continue to increase beyond 10 Gb/s, such interconnects may become necessary to facilitate hardware-level zero-copy across the interconnect.

Previous studies have found high levels of locality in Web server traces [2]. This study confirms those findings. Furthermore, a variety of advanced replacement policies to exploit this locality have been proposed for the Web file cache environment [4], [6]. Other work has considered replacement policies for network file server block caches [24]. The specific policy choices for allocation and replacement are independent of particular implementations of network interface data caching, and any policies could be adopted to improve caching performance.

9 Conclusions

Repeatedly transferring frequently-requested data across a local I/O interconnect leads to an inefficient use of system resources. Furthermore, interconnects scale more slowly than processing power or network bandwidth because of the need for standardization. Caching data directly on a programmable network interface reduces local interconnect traffic on Web servers by eliminating repeated transfers of frequently-requested content. A prototype implementation of network interface data caching reduces PCI bus traffic by 12-61 percent on six Web workloads with only 16 MB caches on two network interfaces. This technique allows application-level performance to scale with more aggressive CPUs and network links beyond the point at which less efficiently utilized local interconnects would become a bottleneck. Such reductions in interconnect traffic require only a few megabytes of DRAM added to the network interface and impose no constraints on the processor on the network interface.

Network interface data caching only requires about 1,000 lines of new and modified code: the addition of five fields to the `mbuf` structure that refer to kernel data buffers, modifications to the `sendfile` system call and `mbuf` manipulation routines, and new code in the device drivers for the network interface. These simple additions to the operating system and 16 MB caches on two network interfaces enable Web server throughput improvements of 6-36 percent for three Web workloads studied, directly resulting from the reduction of data transfers from main memory to the network interface. Therefore, the introduction of a programmable network interface with 16 MB of DRAM would allow existing Web servers to realize this throughput improvement immediately by more efficiently utilizing their local interconnects. Although the prototype uses the FreeBSD `sendfile` system call and a PC-based Web server with a PCI bus, the concepts of network interface data caching are independent of the specific zero-copy I/O mechanism and the specific local interconnect.

Network interface data caching applies in other environments as well, since a variety of systems repeatedly transfer data across the network. Examples include NFS servers, streaming media servers, computation clusters, and network attached storage. While the access locality in each environment will be different, caching data within the network interface is a conceptually simple and practically implementable mechanism for exploiting that locality.

Acknowledgments

The authors thank Alan Cox for suggestions and Erich Nahum for providing the IBM trace as well as comments about this work. This work is supported in part by a donation from Advanced Micro Devices and by the US National Science Foundation under Grant No. CCR-0209174.

References

[1] Alteon WebSystems, *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition,* July 1999.
[2] M.F. Arlitt and C.L. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," *IEEE/ACM Trans. Networking,* vol. 5, no. 5, pp. 631-645, Oct. 1997.
[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE MICRO,* vol. 15, no. 1, pp. 29-36, 1995.
[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Schenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM '99,* vol. 1, pp. 126-134, Mar. 1999.
[5] P. Buonadonna and D. Culler, "Queue Pair IP: A Hybrid Architecture for System Area Networks," *Proc. 29th Int'l Symp. Computer Architecture,* pp. 247-256, May 2002.
[6] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Proc. USENIX Symp. Internet Technology and Systems,* pp. 193-206, Dec. 1997.
[7] P. Druschel and L.L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proc. 14th Symp. Operating Systems Principles (SOSP-14),* pp. 189-202, Dec. 1993.
[8] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE MICRO,* vol. 18, no. 2, pp. 66-76, Mar. 1998.
[9] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual,* 2002.
[10] H.-y. Kim, V.S. Pai, and S. Rixner, "Increasing Web Server Throughput with Network Interface Data Caching," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-X),* pp. 235-250, 2002.
[11] H.-y. Kim, V.S. Pai, and S. Rixner, "Exploiting Task-Level Concurrency in a Programmable Network Interface," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 61-72, June 2003.

[12] K. Kleinpaste, P. Steenkiste, and B. Zill, "Software Support for Outboard Buffering and Checksumming," *Proc. ACM SIGCOMM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm.*, pp. 87-98, Aug. 1995.
[13] Micron Technology, Inc., *Synchronous DRAM 128Mb: x32 SDRAM MT48LC4M32B2*, 2001.
[14] R. Minnich, D. Burns, and F. Hady, "The Memory-Integrated Network Interface," *IEEE MICRO*, vol. 15, no. 1, pp. 11-20, Feb. 1995.
[15] S.S. Mukherjee, B. Falsafi, M.D. Hill, and D.A. Wood, "Coherent Network Interfaces for Fine-Grain Communication," *Proc. 23rd Int'l Symp. Computer Architecture*, pp. 247-258, 1996.
[16] E.M. Nahum, T. Barzilai, and D. Kandlur, "Performance Issues in WWW Servers," *IEEE/ACM Trans. Networking*, vol. 10, no. 2, pp. 2-11, Feb. 2002.
[17] V.S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," *Proc. USENIX 1999 Ann. Technical Conf.*, pp. 199-212, June 1999.
[18] V.S. Pai, P. Druschel, and W. Zwaenepoel, "I/O-Lite: A Unified I/O Buffering and Caching System," *Proc. Third USENIX Symp. Operating Systems Design and Implementation*, pp. 15-28, Feb. 1999.
[19] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," *IEEE MICRO*, vol. 22, no. 1, pp. 46-57, Jan. 2002.
[20] I. Pratt and K. Fraser, "Arsenic: A User-Accessible Gigabit Ethernet Interface," *Proc. IEEE INFOCOM '01*, pp. 67-76, 2001.
[21] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing," *Proc. 2001 ACM/IEEE Conf. Supercomputing (SC2001)*, Nov. 2001.
[22] G. Tyson, M. Farrens, J. Matthews, and A.R. Pleszkun, "A Modified Approach to Data Cache Management," *Proc. 28th Ann. Int'l Symp. Microarchitecture*, pp. 93-103, Dec. 1995.
[23] M. Welsh, A. Basu, and T. von Eicken, "ATM and Fast Ethernet Network Interfaces for User-Level Communications," *Proc. Third Int'l Symp. High Performance Computer Architecture*, pp. 332-342, Feb. 1997.
[24] D.L. Willick, D.L. Eager, and R.B. Bunt, "Disk Cache Replacement Policies for Network Fileservers," *Proc. 13th Int'l Conf. Distributed Computing Systems*, pp. 2-11, May 1993.
[25] K. Yocum and J. Chase, "Payload Caching: High-Speed Data Forwarding for Network Intermediaries," *Proc. 2001 Ann. USENIX Technical Conf.*, pp. 305-317, June 2001.

Hyong-youb Kim is currently a PhD candidate in computer science at Rice University. He received the MS degree in computer science from Rice University in 2003 and the BS degree in computer science from the University of Rochester in 2001. His research interests are Internet network servers, embedded system architectures, and TCP/IP network stack implementations.

Scott Rixner received the BS degree in computer science and engineering in 1995, a MEng degree in electrical engineering and computer science in 1995, and a PhD degree in electrical engineering in 2000, all from the Massachusetts Institute of Technology. Since 2000, he has been an assistant professor in computer science and electrical and computer engineering at Rice University. His research interests include media, network, and communications processing; memory system architecture; and the interaction between operating systems and computer architectures.

Vijay S. Pai received the BSEE degree in 1994, the MS degree in electrical and computer engineering in 1997, and the PhD degree in electrical and computer engineering in 2000, all from Rice University. He joined the faculty of Purdue University in August 2004. Prior to that, he had served as an assistant professor at Rice University and as a senior developer at iMimic Networking. His research interests include computer architecture, networking software, and performance evaluation. He received the US National Science Foundation CAREER award in 2003.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.