Integrated Network Interfaces for High-Bandwidth TCP/IP

Nathan L. Binkert * Ali G. Saidi Steven K. Reinhardt †

Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
University of Michigan
{binkertn,saidi,stever}@eecs.umich.edu

Abstract

This paper proposes new network interface controller (NIC) designs that take advantage of integration with the host CPU to provide increased flexibility for operating system kernel-based performance optimization. We believe that this approach is more likely to meet the needs of current and future high-bandwidth TCP/IP networking on end hosts than the current trend of putting more complexity in the NIC, while avoiding the need to modify applications and protocols. This paper presents two such NICs. The first, the simple integrated NIC (SINIC), is a minimally complex design that moves the responsibility for managing the network FIFOs from the NIC to the kernel. Despite this closer interaction between the kernel and the NIC, SINIC provides performance equivalent to a conventional DMA-based NIC without increasing CPU overhead. The second design, V-SINIC, adds virtual per-packet registers to SINIC, enabling parallel packet processing while maintaining a FIFO model. V-SINIC allows the kernel to decouple examining a packet's header from copying its payload to memory. We exploit this capability to implement a true zero-copy receive optimization in the Linux 2.6 kernel, providing bandwidth improvements of over 50% on unmodified sockets-based receive-intensive benchmarks.

Categories and Subject Descriptors B.4.2 [*Input/Output and Data Communications*]: Interconnections (Subsystems)—Interfaces; C.5.5 [*Computer System Implementation*]: Servers

General Terms Performance, Design, Experimentation

Keywords Network Interfaces, TCP/IP Performance, Zero-copy

1. Introduction

As 10 Gbps Ethernet (10GigE) network components drop in price, they are being more widely deployed in data centers and compute clusters as well as in network backbones. Coupled with the iSCSI protocol, 10GigE is also serving as a storage-area network (SAN) fabric. With 1 Gbps Ethernet effectively the default for current desktop systems, 10GigE connections will be required to avoid contention at any local-area server shared by a reasonable number of these clients [17].

Although users can now plug 10 GigE links into their server systems, getting those systems to keep up with the bandwidth on those links is challenging [14]. Unfortunately for system designers, there is no single bottleneck to high-bandwidth TCP/IP processing that can be easily addressed. Instead, performance losses come from the combination of numerous overheads in interactions between the CPU, memory system, and network interface controller [16, 34].

One very promising and straightforward step to reducing networking overheads is the integration of the NIC on the CPU die. Although not found in current high-performance server systems, CPUs with integrated Ethernet NICs appear in other environments [8, 33], are rumored to be present in some upcoming servers [13], and have been shown to provide substantial performance improvements on server workloads [5]. (See Section 2.1 for further discussion.) Given the trend toward increasing levels of system integration and the ubiquity of Ethernet, we view NIC integration as inevitable in the long term.

While simply integrating a conventional NIC has significant performance benefits, we believe that substantial further opportunities lie in redesigning the NIC to take advantage of its proximity to the host CPU. This paper approaches this redesign process by starting with a minimal on-chip NIC and adding hardware features only as needed. We show that low-latency NIC access allows the OS kernel and device driver to be involved much more directly in the low-level operation of the NIC, opening up the opportunity for optimization of network data handling in the kernel and driver software. In the long term, we believe that the ability to experiment with and deploy significant networking feature and performance upgrades as device driver or kernel releases (rather than, e.g., proprietary firmware updates) will greatly increase the flexibility and performance of network end hosts.

We present and evaluate two on-chip NIC designs. The first, the Simple Integrated NIC (SINIC), strips the device down to its most basic components—a pair of FIFOs—supplemented only by a block copy/checksum unit. All other processing—including the functions typically performed by a DMA engine on even basic conventional NICs—is done in software by the device driver on a general-purpose host CPU. Using detailed full-system simulation, we show that SINIC provides performance comparable to an on-chip conventional DMA-descriptor-based NIC, despite its relative simplicity. Furthermore, though SINIC shifts the responsibility for copying NIC FIFO data from DMA hardware to the device driver, the SINIC system exhibits slightly lower CPU overheads than the system with the DMA-based NIC. This benefit arises because the SINIC driver can copy data directly into kernel internal data structures (`sk_buffs` in Linux), eliminating the overheads of managing a separate DMA descriptor queue and translating between that queue and the kernel internal buffer representation.

* Currently at Hewlett-Packard Labs.

† Also with Reservoir Labs, Inc.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

The key benefit of SINIC is not in providing comparable performance at lower cost, but in creating opportunities for software optimization. To illustrate this potential, we describe a modest set of Linux kernel extensions that allow the protocol stack to defer copying payload data out of the receive FIFO until the packet's protocol header has been processed. If the receiving user process's destination buffer address is known (e.g., because the process has already called `read()`), the packet payload can be copied directly from the NIC FIFO to the user's buffer, achieving true zero-copy operation.

Because the base SINIC design presents a plain FIFO model to software, each packet must be copied out of the FIFO before the following packet can be examined. This restriction significantly limits packet-level parallelism when the deferred copy technique is applied. Our second NIC design removes this restriction by adding virtual per-packet control registers to the SINIC model. This extended interface, called V-SINIC, enables overlapped packet processing in both the transmit and receive FIFOs. With V-SINIC, the device driver can initiate processing on newer packets even while older packets wait in the FIFO for their destination memory address to be determined. Again using full-system simulation, we show that V-SINIC with our zero-copy extensions implemented in the Linux 2.6 kernel provides bandwidth improvements of over 50% on an unmodified sockets-based receive-intensive micro-benchmark.

The remainder of the paper begins with a qualitative case for simple integrated network interfaces, including comparisons to other approaches such as TCP offload engines (TOEs). We follow with a detailed description of our SINIC and V-SINIC designs and a discussion of related work. We then describe our evaluation methodology and present our results. Finally we offer conclusions and a discussion of future work.

2. The Case for Simple Network Interfaces

In this section, we provide qualitative arguments for architecting a simple, low-level network interface for high-bandwidth TCP/IP servers. At the lowest level, a network interface controller is a pair of FIFOs (transmit and receive) plus some control information. The only components beneath this interface are the medium access control (MAC) and physical interface (PHY) layers, which are dependent on the physical interconnect. Our basic proposal is to expose these FIFOs directly to kernel software. Injecting programmability at the lowest possible layer allows a common hardware platform to adapt to a variety of external networks and internal usage models. Just as software-defined radio seeks to push programmability as close to the antenna as possible, we seek to push programmability as close to the wire as possible to maximize protocol flexibility.

We first discuss the case for integrating the NIC on a processor die, a prerequisite for our simple NIC structure. We then contrast our approach with two alternatives: current conventional NIC designs and the TCP offload engine approach that represents a contrasting vision of future NIC evolution.

2.1 The Case for NIC Integration

A high-bandwidth NIC requires significant amounts of closely coupled processing power in any design. The key enabler for our simple NIC approach is the ability to have one or more general-purpose host CPUs provide that power. This coupling is easily achieved by integrating the NIC on the same die as the host CPU(s). Although integrated Ethernet NICs are not found on high-performance servers today, there are numerous examples in other environments, including some embedded network processors [8] and BlueGene/L [33], whose nodes use an integrated NIC for I/O. Given available transistor budgets, the potential performance benefits [5], and the importance and ubiquity of high-bandwidth Ethernet, NIC integration is an obvious evolutionary step in the high-performance domain as well. Future revisions of Sun's Niagara product line are rumored to include one or more integrated 10GigE NICs [13].

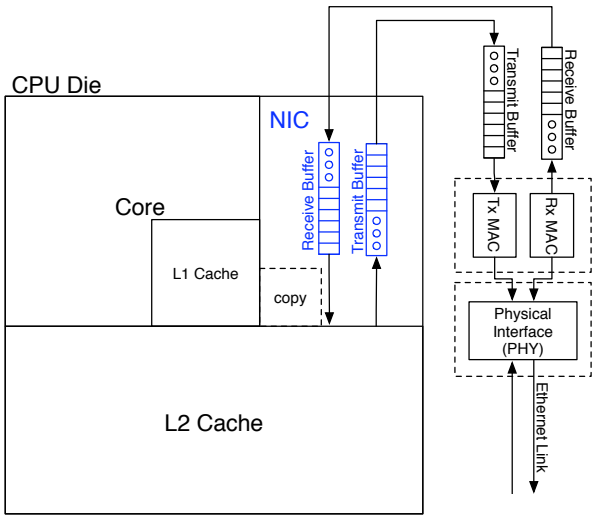

Figure 1. High-level block diagram of SINIC

For our simple NIC, only one end of each FIFO and its associated control logic need be integrated on the die. As shown in Figure 1, additional FIFO buffer space and the physical link interface (PHY) can be off-chip. The on-chip to off-chip connection could either use dedicated pins or be multiplexed onto one of the CPU's existing interfaces (e.g., a HyperTransport link). If the memory controller is also on the CPU die, as is increasingly likely in the future, then network data must be routed onto the die in a conventional DMA system as well, so the FIFO interface does not incur any additional pin bandwidth. The off-chip PHY also enables a single design with an integrated NIC to support different physical media (e.g., copper or fiber). With the addition of a simple off-chip mux/demux circuit, a single FIFO could also support multiple links (e.g., a 40 Gbps FIFO connected to four 10 Gbps links).

2.2 Simple Versus Conventional NICs

Conventional Ethernet NICs are designed to reside on a standard I/O bus (e.g., PCI) that is physically distant from and clocked much more slowly than the CPU, such that uncached accesses to device registers may require thousands of CPU cycles [5]. Due to these overheads, requiring the CPU to interact frequently with the device is impractical. A conventional NIC thus uses DMA to manage large main-memory-based FIFOs, copying data to and from these structures into its on-board FIFOs as needed. To provide flexibility in memory allocation, these main-memory FIFOs are non-contiguous, represented by lists of memory-resident DMA descriptor data structures. Each descriptor contains the address and length of a contiguous buffer.

To transmit a packet, the device driver creates a DMA descriptor for each of the internal kernel buffers that make up the packet (often one for the protocol header and one for the payload), writes the DMA descriptors to the in-memory transmit queue, then writes a NIC control register to alert it to the presence of the new descriptors. The NIC then performs a DMA read operation to retrieve the descriptors, a DMA read for each data buffer to copy the data into the NIC-resident hardware FIFOs, then a DMA write to mark the

descriptors as having been processed. The device driver will later reclaim the DMA descriptors and buffers.

Receive operations are similar, except that the device driver preallocates empty buffers and places corresponding DMA descriptors on a queue for the NIC to fill with received packets. After each buffer is filled, the NIC marks its descriptor accordingly and interrupts the CPU. The device driver then processes the filled buffers, converting them to internal kernel format, and passes them to the kernel's protocol stack.

Though this process of fetching, processing, and updating DMA descriptors is conceptually simple, it incurs a non-trivial amount of memory bandwidth and processing overhead, both on the NIC and in the device driver. Willmann et al. [39] analyzed a commercial 1 Gbps Ethernet NIC that implements DMA in firmware and determined that an equivalent 10 Gbps NIC must sustain 435 MIPS to perform these tasks at line rate. Note that, other than possibly calculating checksums, this computational effort includes *no* inspection or processing of the packets whatsoever.

In contrast, the reduced latency afforded by on-chip integration allows the NIC to operate without the expanded buffer space provided by the DMA descriptor queues. A simple NIC that directly exposes the hardware FIFOs to software does not require DMA descriptors at all, avoiding all descriptor management overhead. A further advantage of the simple NIC approach is that the payload data buffer used on receive can be selected dynamically by the device driver based on the packet header, unlike the DMA descriptor model where receive buffers must be populated by the driver in advance.[1] In Section 3.3, we describe a set of kernel modifications that take advantage of this feature to provide true zero-copy receives—where data is copied directly from the NIC FIFO into the user's destination buffer—for unmodified socket-based applications.

2.3 Simple NICs Versus TCP Offload Engines

An alternative to coupling the NIC FIFOs with the central processing unit is to leave the NIC on an I/O bus and add processing power to it. An extreme example of this approach is a TCP offload engine (TOE), in which the NIC itself is responsible for most or all of the TCP and IP protocol processing [1, 7, 18].

A key disadvantage of TOEs is a lack of flexibility. Protocol implementations are not accessible to system programmers and are not easily changed. As existing protocols evolve and new protocols are developed, users must wait not only for protocol support from their operating system but also from their NIC vendor, and for both of these to happen in a coordinated and compatible fashion. Although the Internet seems stable, new protocols are not uncommon; consider IPv6, IPSec, iSCSI, SCTP, RDMA, and iSER (iSCSI Extensions for RDMA). This situation will be particularly problematic if the update in question is a fix to a security vulnerability rather than a mere performance issue.

A corollary of this lack of flexibility is that TOEs are not easily customized for or tightly integrated with particular operating systems. In fact because of the overwhelming number of changes to the operating system code and the short-circuiting of other network stack features (e.g. netfilter) it is extremely unlikely that TOE support will make it into the mainline Linux kernel [10, 27]. In contrast, the kernel support for SINIC fits in the standard driver framework, and requires no changes to the kernel network stack. Because SINIC's intelligence lies within the driver and not in device firmware, SINIC has direct access to kernel code and data structures, a feature that is critical to achieving our zero-copy extensions in Section 3.3.

Other arguments against this direction include the inability of TOEs to track technology-driven performance improvements as easily as host CPUs [28, 34], that TOEs dedicate processing power to networking that cannot be used for other purposes when the network is idle, and that TOEs provide significant speedups only under a limited set of workload conditions [37].

The TOE trend of pushing more intelligence out to the NIC brings to mind Myer and Sutherland's "wheel of reincarnation" [30]. They observe that a peripheral design tends to accrue more and more complexity until it incorporates general-purpose processing capabilities, at which point it is replaced by a general-purpose processor and a simple peripheral, corresponding to a full turn of the wheel. Our SINIC design can be seen as the next step past TOEs around the wheel of reincarnation.

3. The Simple Integrated Network Interface Controller (SINIC)

This section describes our integrated NIC designs in detail, along with their associated device drivers. We begin with our simple integrated NIC (SINIC). SINIC by itself is not intended to provide higher performance than a similarly integrated conventional NIC. Instead, its design provides comparable performance with added flexibility and reduced implementation complexity. Because SINIC adheres to a strict FIFO model, it limits the amount of packet-level parallelism the kernel can exploit. We then describe V-SINIC, an extension to SINIC that enables overlap in packet processing by providing virtual per-packet registers. Finally we describe how we use V-SINIC to implement zero-copy receives in Linux 2.6.

3.1 Base SINIC Design

As discussed in Section 2, conventional NICs provide a software interface that supports the queuing of multiple receive and transmit buffers via a DMA descriptor queue. Due to its close proximity to the host CPUs, SINIC is able to achieve comparable performance without a queuing interface and without scatter/gather DMA. Instead, SINIC consists of little more than a pair of FIFOs and a pair of copy engines.[2] The copy engines include adders to generate packet checksums during the copy. The only additional logic is a set of comparators on the RX side to identify the incoming packet protocol and verify the packet's checksum. The system-level view of SINIC is shown in Figure 1. The receive half of SINIC is detailed in Figure 2; the transmit half is nearly identical.

The core of the SINIC interface consists of four memory-mapped registers: RxData, RxDone, TxData, and TxDone, illustrated in Figure 3. The CPU initiates a copy operation from the receive FIFO to memory by writing to the RxData register, and conversely from memory to the transmit FIFO by writing to TxData. In both cases, the address and length of the copy are encoded into a single 64-bit data value written to the register. The TxData value encodes two additional bits. One bit (MORE) indicates whether this copy terminates a network packet; if not, SINIC will wait for additional data before forming a link-layer packet. The other bit (CSUM) enables SINIC's checksum generator for the packet.

SINIC operates entirely on physical addresses. Because it is designed for kernel-based TCP/IP processing, it does not face the address translation and protection issues of user-level network interfaces. Instead, the driver code simply calls internal kernel functions to translate virtual addresses to physical addresses when necessary. Because a translation is only used for a brief period (while the copy

[1] Some higher-end NICs provide multiple receive queues and a hardware packet classification engine that selects a queue based on protocol header matching rules, but these NICs are more complex and limited both in the number of queues and in the number of matching rules.

[2] In our simulations, these engines share a single L2 cache port with the L1 caches, so only one can be transferring at any given time.

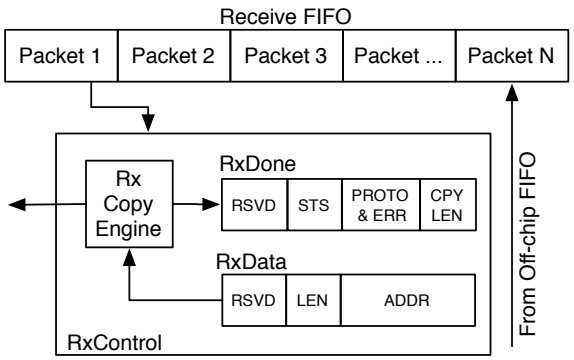

Figure 2. SINIC block diagram (receive-side only)

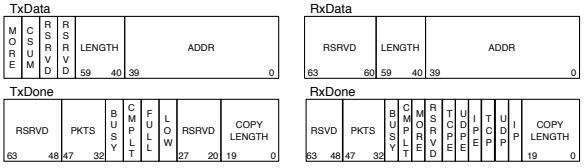

Figure 3. Principal SINIC registers

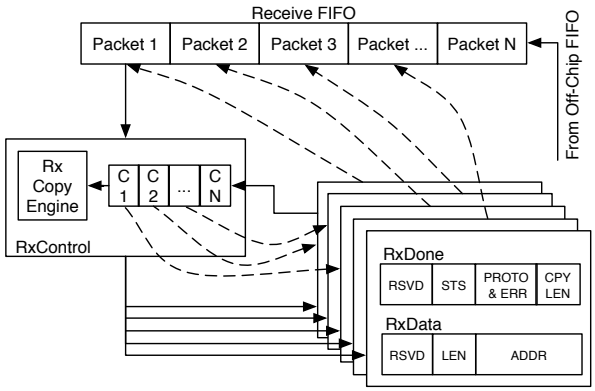

Figure 4. V-SINIC block diagram (receive-side only)

engine is active), page invalidations on active pages can be handled by delaying the response to TLB shootdown requests until the copy completes (although we did not implement this feature).

The RxDone and TxDone registers provide status information on their respective FIFOs. Each register indicates the number of packets in the FIFO, whether the associated copy engine is busy, whether the last copy operation initiated on the FIFO completed successfully, and the actual number of bytes copied. (This last value is useful as it allows the driver to provide the allocated buffer size as the copy length to the receive FIFO and rely on SINIC to copy out only a single packet even if the packet is shorter than the buffer.) TxDone also indicates whether the transmit FIFO is full. RxDone includes several additional bits. One bit indicates whether there is more data from the current packet in the FIFO. Another set of bits indicates whether the incoming packet is an IP, UDP, or TCP packet, and whether SINIC's calculated checksum matched the received packet checksum for each protocol (the IPE, UDPE, and TCPE bits in Figure 3).

Because SINIC implements a single copy engine per FIFO, the CPU must let each copy complete before initiating another copy. Individual buffer transfers are relatively fast, so the driver just waits when it needs to perform multiple copies. Rather than busy-waiting on RxDone or TxDone, SINIC enables more efficient synchronization through two additional status registers, RxWait and TxWait. These registers return the same status information as RxDone and TxDone, respectively, but a load to either of these registers is not satisfied by SINIC until the corresponding copy engine is free. Thus a single load to RxWait replaces a busy-wait loop of loads to RxDone, reducing memory bandwidth and power consumption.

In addition to these six registers, SINIC has interrupt status and mask registers and a handful of configuration control registers.

Like a conventional NIC, but unlike a TOE, SINIC interfaces with the kernel through the standard network driver layer. SINIC's device driver is simpler than conventional NIC drivers because it need not deal with allocation of DMA descriptors, manage descriptors and buffers (e.g., reclaim completed transmit buffers), nor translate the kernel's buffer structures (e.g., Linux `sk_buffs`) into the NIC's DMA descriptor format. When a packet must be transmitted, the device driver simply loops over each portion of the packet buffer; for each portion, it initiates a transmit with a programmed I/O (PIO) write to TxData, then busy waits on the result with a PIO read to TxWait. For the final portion of the packet, the driver does not wait on the copy to complete; instead, it allows the copy to overlap with computation, and verifies the engine to be free before initiating the next packet transmission.

3.2 Virtualizing SINIC for Packet-Level Parallelism

As long as each packet is copied to or from memory in its entirety before the next packet is processed, a single blocking copy engine per FIFO is adequate. However, there are situations—such as the zero-copy optimization described in the following section—where it is useful to begin processing a packet before the preceding packet is completely copied into or out of the FIFO. This feature is particularly desirable for chip multiprocessor systems, where packet processing can be distributed across multiple CPUs.

We extend the SINIC model to enable packet-level parallelism by providing multiple sets of RxData, RxDone, TxData, and TxDone registers for each FIFO and dynamically associating different register sets with different packets. We call this the *virtual SINIC (V-SINIC)* model, as it gives each in-process packet its own virtual interface. The receive half of V-SINIC is depicted in Figure 4. For brevity, we will refer to a single set of virtual per-packet registers as a *VNIC*. V-SINIC still has only one copy engine per direction, but each engine is multiplexed dynamically among the active VNICs. V-SINIC supports one outstanding copy per VNIC; once a copy is initiated on a VNIC, that VNIC will be marked busy until it acquires the copy engine and completes the copy.

Although the V-SINIC extensions to both the receive and transmit FIFOs are conceptually similar, they differ slightly in details and significantly in usage. On the transmit side, V-SINIC is used to allow concurrent lockless access from multiple CPUs in a CMP. Each CPU is statically assigned a VNIC. If two CPUs attempt to transmit packets simultaneously, V-SINIC's internal arbitration among the VNICs will serialize the transmissions without any synchronization in software. To avoid interleaving portions of different packets on the link, once a VNIC acquires the copy engine it maintains ownership of the engine until a complete packet is transferred, even across multiple individual copy requests (e.g., for the header and payload). This policy applies to the transmit FIFO only; as will be described shortly, the receive FIFO is specifically designed to

allow interleaving of headers and payloads from different packets as they are copied out.

On the receive side, V-SINIC enables two optimizations. First, the driver can pre-post buffers by initiating copy operations to different buffers on multiple VNICs, even if the receive FIFO is empty. As packets arrive, the copy operations are triggered on each VNIC in turn. The driver then uses the per-VNIC RxDone registers to determine the status of each packet.

The second receive-side optimization is deferred payload copying. Because VNICs are bound to packets, once part of a packet is received via a particular VNIC, the remaining bytes of that packet can only be retrieved by a subsequent copy request to the same VNIC. For a given packet, the low-level driver can copy just the header to memory, examine the header, then hand off the VNIC to another CPU for further processing. At some later point in time, the other CPU can initiate the copy of the packet payload out of the FIFO. In the interim, the driver continues to process additional headers from subsequent packets using other VNICs. If packets are quickly copied into kernel buffers, the additional parallelism exposed by deferred copying is minimal. However, the deferred copy capability is critical for our implementation of zero-copy receives described in the following section.

3.3 Implementing Zero-Copy Receives on V-SINIC

The overhead of copying packet data between kernel and user buffers is often a significant bottleneck in network-intensive applications. In the normal case, an entire packet is DMAed into a driver-provided buffer (e.g. an `sk_buff` in the case of the Linux kernel). Later, when the packet's destination process is determined, the data is copied out of the buffer into the receiving process' memory (unless the kernel itself is the recipient). This copy can be avoided by moving data directly from the NIC FIFO into the user buffer. This "zero-copy" behavior is practically impossible to achieve with a conventional DMA-descriptor-based NIC, as the driver cannot inspect the packet's header to decide where it needs to go until the entire packet has been copied to a pre-allocated DMA buffer.

V-SINIC's deferred-copy capability enables a straightforward implementation of zero-copy receives in the Linux 2.6 kernel. We added a flag to Linux's `sk_buff` structure to indicate that the referenced data is resident in the V-SINIC FIFO, and a field to specify the VNIC index in that case. We also modified the two kernel functions used to copy `sk_buff` contents to recognize this encoding (`skb_copy_bits()` and `skb_copy_datagram_iovec()`, which copy `sk_buff`s to kernel and user buffers, respectively). When either of these functions is called and the `sk_buff` indicates that the data is still in the V-SINIC FIFO, the code instructs the VNIC to copy the data directly from the FIFO into the user buffer, thus avoiding the intermediate copy described above.

4. Related Work

On-chip integrated network interfaces have appeared before in the context of fine-grain massively parallel processors. Henry and Joerg [21] investigated a range of placement options, including on- and off-chip memory-mapped NIs and a NI mapped into the CPU's register file. Other machines with on-chip network interfaces include the J-Machine [11], M-Machine [15], and *T [32] research projects, and IBM's BlueGene/L [33]. Mukherjee and Hill [29] also argue for tighter coupling between CPUs and network interfaces for storage-area and cluster networks. They focus placing the NIC in the coherent memory domain but not on physical integration with the CPU. SHRIMP [6] also uses a memory-mapped copy engine for their user-level DMA, though in conjunction with complex mechanisms to allow protected user-level access via virtual addresses. In all of these cases, the primary goal is low user-to-user latency using lightweight protocols and hardware protection mechanisms. In contrast, TCP/IP processing has much higher overhead and practically requires kernel involvement to maintain inter-process protection. Our work continues in the spirit of this research, but focuses on optimizing the NIC interface for the needs of the kernel's TCP/IP stack. SINIC's close coupling of a processor to receive and transmit FIFOs is similar to the coupling seen in many network processors such as Intel's IXP [25]. Our work differs in that we expose the FIFOs directly to kernel software running on the primary host CPU rather than firmware running on dedicated packet engines.

In the TCP/IP domain, a few other groups have investigated alternatives to offloading. Our earlier work [5] investigated the benefit of integrating a conventional DMA-based NIC on the processor die, but did not consider modifying the NIC's interface to exploit its proximity to the CPU. Intel announced an "I/O Acceleration Technology" (I/OAT) initiative [26] that explicitly discounts TOEs in favor of a "platform solution" [20]. Intel researchers proposed a "TCP onloading" model in which one CPU of an SMP is dedicated to TCP processing [34]. Our SINIC model is complementary to this approach: the dedicated CPU would likely benefit from a closely coupled flexible NIC. Another Intel paper describes "direct cache access" I/O [24], in which incoming DMA data from an external NIC is pushed up into the CPU's cache. Placing incoming network data in the on-chip cache is natural when the NIC is on the same chip, and we see similar benefits from this effect.

Zero-copy (more accurately single-copy) receives have been implemented in other contexts [9]. The most common technique is *page flipping*, where the buffer is copied from the kernel to the user address space by remapping a physical page. Trapeze [19] provided zero-copy in this manner, but has limitations such as requiring page-size MTUs (much larger than the Internet standard 1500 bytes). The header and payload separation done by Trapeze is very similar to the zero-copy mechanism that we use to demultiplex packets, though we do not require that the extra intelligence for this separation exist in the NIC. In addition, the required page-table manipulations, while faster than an actual page copy, are not quick in an absolute sense. Zero-copy behavior can also be achieved using "remote DMA" (RDMA) protocol extensions, where the receiving application pre-registers receive buffers in such a way that the NIC can identify them when they are referenced in incoming packets. In addition to requiring a sophisticated NIC, RDMA is a significant change to both applications and protocols, and requires support on both ends of a connection. V-SINIC does not preclude the use of RDMA—in fact, V-SINIC's flexibility allows adding an efficient RDMA protocol implementation solely as a driver upgrade, with no additional hardware needed. However, many of RDMA's benefits are realized in V-SINIC's zero-copy optimization without requiring modified protocols or applications.

Our V-SINIC approach is most closely related to that of Afterburner [12], an experimental NIC that combined significant onboard buffering with a modified protocol stack such that copying packet payloads off of the NIC could be deferred until the destination user buffer was known. Although Afterburner reduced copy overhead much like V-SINIC, its low-level mechanisms were different. It exposed the NIC buffer memory to the CPU as a single, flat memory region, shared with the NIC by using a three-port video RAM. This buffer was managed in software, much like a conventional set of DMA buffers. Copies to and from the buffer used a special optimized software copy routine. V-SINIC differs by using single-ported buffer memory with a specialized control-register interface. The buffer is managed in hardware as a FIFO, with additional logic for copying data, managing individual packet buffers, and multiplexing between requests. V-SINIC's specialized interface paves the way for other optimizations not readily supported by Afterburner, such as an efficient interface for virtual machines or enabling direct user access in a secure way.

V-SINIC also differs from Afterburner in its placement on the CPU die. (Afterburner plugged into systems via the graphics card slot.) As a result, V-SINIC can place data directly in the on-chip cache hierarchy. V-SINIC also needs less buffer space despite supporting higher bandwidths, because the amount of buffering required is proportional to the product of the network bandwidth and the CPU/NIC latency. Although Afterburner was relatively closely coupled to the CPU, it had 1 MByte of on-board buffering for a 1 Gbps network. The extremely low latency afforded by on-chip integration allows SINIC to support a similar technique on a 10 Gbps network with substantially less buffering. Our simulated implementation uses up to 380 KB of space in the receive FIFO—256 VNICs times 1514 bytes per packet—but, as shown in Section 6, performance may not suffer significantly until the buffer space drops below 100 KB.

5. Methodology

We evaluated the SINIC design by running TCP/IP-based micro- and macrobenchmarks on an appropriately modified full-system simulator. The following subsections discuss the simulation environment and the benchmarks in turn.

5.1 Simulation Environment

Conventional architecture simulators cannot adequately model the performance of network-intensive workloads for two reasons. First, much of the workload's execution time is not spent running application code, but rather kernel code such as device drivers and the TCP/IP stack. As a result, simulators that merely emulate system calls do not provide meaningful results for network benchmarks. Second, achieving accurate timing results for I/O-intensive workloads requires detailed modeling of the main memory and I/O subsystems and relevant I/O devices, such as the NIC. Typical architecture simulators focus on modeling of the CPU microarchitecture and first-level caches, but little beyond this scope.

To address these issues, our research group developed M5 [4], a simulator targeted at research in TCP/IP network I/O. M5 is a full-system simulator, modeling the Alpha Tsunami platform with enough fidelity to boot an unmodified Linux 2.6 kernel and run Alpha PALcode. M5 uses a simple configurable bus model to provide event-driven timing models for all levels of the memory hierarchy, including the main-memory and I/O busses. I/O devices and their interactions with the memory system are modeled in detail, including both programmed I/O and DMA accesses. M5's performance models have been validated against a real Alpha Tsunami system (a Compaq XP1000) [35].

For our conventional NIC, we use M5's model of a National Semiconductor DP83820 [31] Gigabit Ethernet device and scale its performance to 10 Gbps. The model is accurate enough to support the standard Linux driver for this device. However, our model fixes a bug in the real hardware that prevents the NIC from issuing DMAs to unaligned addresses, and we use a modified driver that does not contain a workaround for this hardware bug. We use a fixed delay in the NIC to bound the interrupt generation rate to once per 10 μs.

All our experiments use a two-system client-server configuration. In each case, only one system is of interest, while the other merely serves as a stressor. The system of interest, whether the client or the server, is modeled in detail. The stressor is modeled purely functionally—using a 1 CPI CPU and a perfect 1-cycle memory system—so as not to be a bottleneck. The simulated systems are connected directly using a lossless 10 Gbps Ethernet link. Table 1 lists the other parameters we used for the detailed system simulation. Since the memory system we are modeling is similar to that of an Opteron system, we configured the latency memory, bus bridges and peripheral devices to match numbers measured on an AMD Opteron server.

Frequency	2 GHz or 4 GHz
Fetch Bandwidth	Up to 4 instructions per cycle
Branch Predictor	Hybrid local/global (e.g. EV6)
Instruction Queue	Unified int/fp 64 entries
Reorder Buffer	128 Entries
Execution BW	4 insts per cycle
L1 Icache/Dcache	128KB, 2-way set assoc, 64B blocks, 16 MSHRs. 1 cycle inst hit; 3 cycle data hit.
L2 Unified Cache	Size varies, 8-way set assoc. 64B block size, 25 cycle latency, 40 MSHRs.
L1 to L2	64 bytes per CPU cycle
L2 to Memory	4 bytes per CPU cycle
HyperTransport	8 bytes, 800 MHz
Main Memory	50ns

Table 1. Simulated System Parameters

5.2 Benchmarks

We evaluate network performance using receive and transmit tests from the Netperf microbenchmark suite and two macrobechmarks (a modified SPECweb99 and iSCSI).

Netperf [22] is a network microbenchmark suite developed at Hewlett-Packard. We use the TCP stream benchmark and TCP maerts benchmark. In either case, the client (the system under test) connects to a server and sends or receives data as quickly as possible over a single TCP connection. The stream benchmark is a transmit benchmark, setting up a socket and calling `send()` in a tight loop. Normally this call returns as soon as the data is copied out of the user's buffer. However, if the kernel socket buffer is full, the call will block until space is available. This blocking behavior makes the benchmark self-throttling. The maerts benchmark is the reverse: the client receives data as quickly as possible from the server. Naturally, it loops on `receive()` instead. In general, the time spent executing the benchmarks code is minimal, and most of the CPU time is spent in the kernel driver managing the NIC or processing the packet in the TCP/IP stack.

SPECweb99 [38] is a popular benchmark that is used for evaluating the performance of web servers. The benchmark simulates multiple users accessing a combination of static and dynamic content using HTTP 1.1 connections. In our simulations we used Apache 2.0.52 [2] with the mod_specweb99 CGI scripts. These scripts replace the reference implementation with a more optimized version written in C, and are frequently used in the results on the SPEC website.

The standard SPECweb99 client is fairly heavyweight yet each connection provides only a modest load on the server; the standard SPECweb99 score is based on the maximum number of simultaneous clients that the server can support while meeting some minimum bandwidth and response time guarantees. Because of the resources required to run a sufficient number of clients to generate multiple Gbps of load, the client is not well suited for a simulation environment. Since we are not concerned with the SPECweb99 score attainable by the machine under test, but rather are simply interested in the performance characteristics of a web server workload, we chose to use a different client based on the Surge traffic generator [3], Our client preserves the same statistical request distribution as the original client, but is able to scale its performance

Benchmark	Warm-up	Sampling
Netperf Single-stream	100M	50M
Netperf Multi-stream	500M	100M
SPECweb99	1B	400M
iSCSI	1B	100M

Table 2. Number of Instructions Simulated

up to a point that can saturate the server while still remaining feasible to simulate.

iSCSI [36] is a new standard implementing the SCSI protocol on top of a TCP/IP connection. This protocol allows an initiator (client) to access a target (server) much like it would access a local SCSI device. Because of its use of TCP/IP as a connection layer protocol and Ethernet as a link layer protocol, iSCSI promises to be much cheaper than previous network storage systems (e.g., FibreChannel).

In our tests we used the Open-iSCSI initiator and the Linux iSCSI Enterprise target. Because we are concerned with network rather than disk I/O performance, the target does not have a real I/O backing store, but instead returns data immediately. On top of the iSCSI client we run a custom benchmark that uses Linux's asynchronous I/O (AIO) facilities to continuously maintain multiple outstanding reads to the iSCSI disk in flight. As soon as a read completes, a new location to read is selected and is issued to the disk. We benchmark both the target and the initiator.

For the experiments in this paper we used a 1500 byte maximum transfer unit (MTU) as it is the standard on the Internet today. Although increasing the MTU is reasonable in a dedicated environment, larger MTUs are never used for commodity traffic on the Internet.

Running these workloads to completion in simulation is infeasible due to the significant slowdown incurred. Thus we turn to standard fast-forwarding, warm-up and sampling techniques to gather data on different NIC configurations. Our benchmarks have been modified to inform the simulator when they reach a stable point in their execution. We initially run the benchmarks in a purely functional simulation until they reach this point, at which time the simulator checkpoints all program state. Experiments are run by restoring this checkpointed state, warming up the caches and TLB, and then switch to a detailed timing model to gather statistics. For our experiments our warm-up and simulation periods were as listed in Table 2. The warm-up period is of a lower effective performance than the detailed simulation so that the TCP protocol can adjust quickly to the effective performance change from simple to detailed simulation [23].

6. Results

In this section, we investigate how SINIC and V-SINIC impact overall performance, cache performance, and CPU utilization across our benchmarks. In addition, we investigate the zero-copy optimization that V-SINIC makes possible, and the impact of the number of VNICs on the zero-copy optimization.

First, we look at the performance of SINIC when compared to a conventional NIC with both an on-chip and off-chip attachment. The off-chip conventional NIC is attached to a PCI Express-like bus, which is in turn connected to the CPU by way of an I/O bridge chip using a HyperTransport-like interconnect. We also investigate the performance of the same conventional NIC in an on-chip configuration with DMA engine access to the on-chip last level cache. Comparing the on-chip CNIC to SINIC isolates the impact of the SINIC design itself.

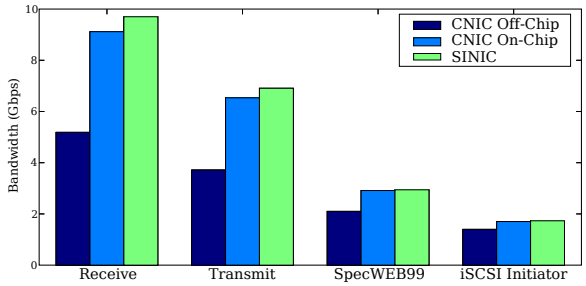

Figure 5. Achieved bandwidth for a conventional NIC (CNIC) vs. SINIC (8MB Cache)

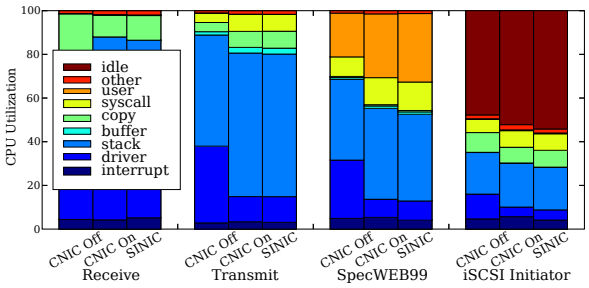

Figure 6. CPU utilization breakdown for a conventional NIC (CNIC) vs. SINIC (8MB Cache)

Figure 5 shows the performance of these configurations. Comparing the two conventional NIC (CNIC) attachments clearly shows that on-chip integration of the NIC in the system increases performance significantly. There are two main reasons for this. First, the CPU's device register accesses have much lower latency. Second, the device can place incoming DMA data directly into the on-chip cache, reducing the amount of memory traffic and cache misses, often dramatically [5, 24].

Though simplicity is the goal of SINIC, we see that SINIC slightly outperforms the more complex CNIC attached at the same position. SINIC's simplicity leads directly to its increased performance. As described in Section 2.2, the CNIC implements a DMA descriptor mechanism for managing buffer transfers to and from the network. SINIC avoids the overhead of managing DMA descriptors at the cost of requiring the CPU to initiate data copies directly. The close attachment of an on-chip NIC to the CPU makes the latency tolerance provided by the DMA descriptors unnecessary, making SINIC's direct approach more efficient. SINIC's increased CPU overhead due to managing the FIFOs is more than offset by the removal of the the descriptor management code from the device driver. As a result, SINIC provides similar performance to the on-chip CNIC even for complex CPU-bound workloads such as SPECweb99.

In Figure 6 we show a breakdown of how the CPU spent its time on each platform. Note the reduction in driver time seen when comparing the off-chip NIC with the on-chip attachments. This reduction is a direct result of the lower latency of device register accesses for the on-chip NICs, providing more CPU time for processing the data. The increased percentage of stack and user time seen in the on-chip configurations correlates directly with the increased bandwidth provided by these platforms.

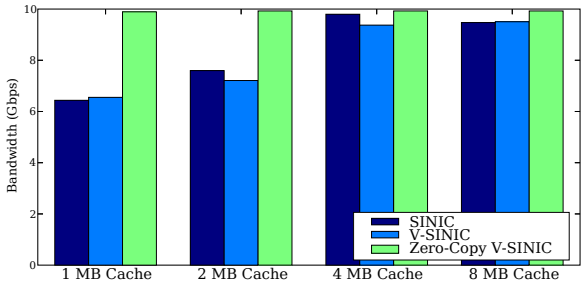

Figure 7. Achieved bandwidth for the receive micro-benchmark using the zero-copy optimization

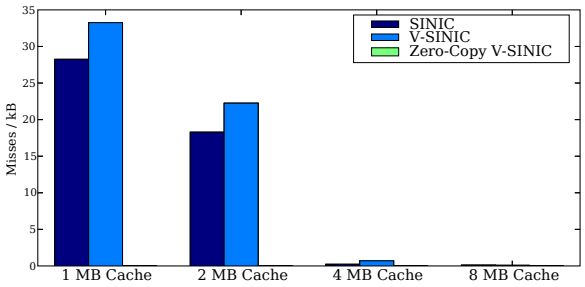

Figure 8. Cache miss rate in misses per kilobyte of data transferred for the receive micro-benchmark using the zero-copy optimization

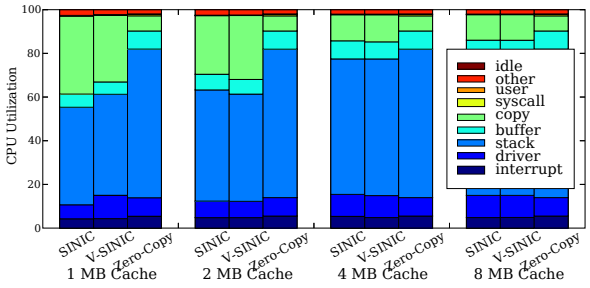

Figure 9. Breakdown of CPU utilization for the receive micro-benchmark using the zero-copy optimization

Figure 7 compares the performance of SINIC with V-SINIC with and without the zero-copy optimization on the receive microbenchmark, where we expect copy elimination to have the greatest impact. Repeating the result shown in Figure 5, the non-zero-copy SINIC is able to saturate the network with a large (4-8 MB) L2 cache. In this case, the direct cache placement of incoming network data makes the buffer copies efficient cache-to-cache operations. However, for smaller L2 cache sizes, the SINIC network buffers overflow into main memory. The overhead of the resulting DRAM-to-cache buffer copies causes significant bandwidth degradation (a 40% reduction down to 6 Gbps with a 1 MB cache). Without the zero-copy optimization, V-SINIC provides similar performance characteristics, though at a slightly reduced performance level due

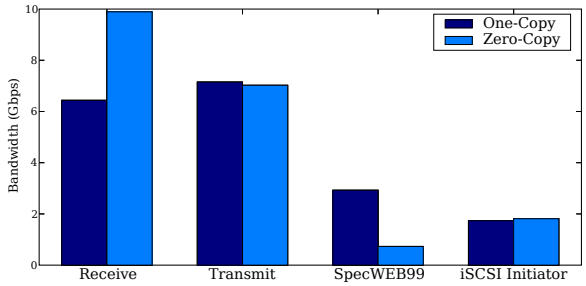

Figure 10. Achieved bandwidth with an aggressive zero-copy optimization (1MB Cache)

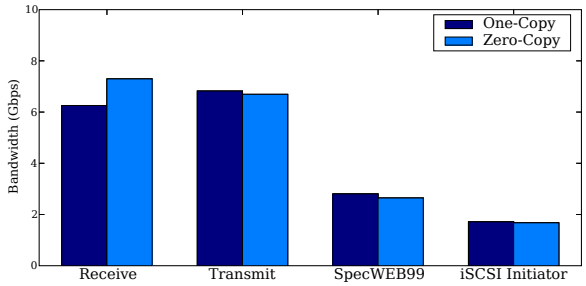

Figure 11. Achieved bandwidth with a conservative zero-copy optimization (1MB Cache)

to additional overheads in the device driver.[3] In contrast, the zero-copy optimization eliminates the buffer copy entirely, making performance insensitive to the cache size, and allowing the system to saturate the network in every configuration.

Zero-copy V-SINIC is useful whenever network buffers do not fit in the on-chip cache, which can occur even with large caches. Our microbenchmark touches very little data other than the network buffers, and the latency of our simulated network was extremely low, meaning that network buffer space was relatively small. As a result, we see a benefit from zero copy only on fairly small cache sizes. Benchmarks with larger non-network working sets would leave less room in the cache for network buffers, and larger network delays will increase the amount of network buffering required. Both of these situations are likely in real systems, leading to zero-copy benefits even at larger cache sizes.

Figure 8 shows that the number of cache misses per kilobyte transferred correlates strongly with network performance. While SINIC's cache data placement coupled with a sufficiently large cache drives the cache miss rate of the buffer copies to zero, the zero-copy V-SINIC model incurs practically no cache misses, regardless of cache size, because network data is read directly from the FIFO rather than from memory-resident kernel buffers. Figure 9 shows that, as a result of dramatically reduced copy time, the zero-copy V-SINIC system can spend more time in the TCP/IP stack processing packets. The copy time shown for the zero-copy V-SINIC case corresponds to the time the CPU spends setting up and waiting for the V-SINIC copy engine to copy data from the FIFO to the user buffer.

[3] We expect that it will be possible to reduce or eliminate these overheads with more careful driver optimization.

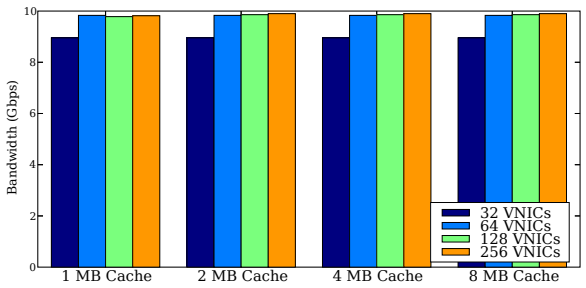

Figure 12. Performance of the receive micro-benchmark with varied numbers of VNICs.

Figure 10 shows the performance of the zero-copy optimization across our set of benchmarks. These results use a 1MB cache, where the impact of the optimization is greatest (see Figure 7). Because the optimization only applies to received packets, the transmit microbenchmark is unaffected. The iSCSI benchmark also sees no impact. However, the SPECweb99 benchmark suffers a noticeable performance degradation. We discovered that SPECweb99 has a very long latency from when a packet is received and processed by the kernel until it is accessed by a `read()` call from the application. This latency arises because Apache blocks in the `select()` system call (rather than pre-posting a buffer and blocking in `read()`). The effect is exacerbated by the large number of threads Apache uses, such that even once the process is notified via `select()` that a packet has arrived, there may be a significant delay before the relevant thread is scheduled for execution. The impact of this latency is that the V-SINIC FIFO fills up with packets waiting for `read()` calls to provide their final destination buffer address.

In order to cope with the long latency between packet reception and `read()`, we have implemented a more conservative zero-copy optimization that tries to only use zero-copy when the receive FIFO is not full. Figure 11 shows the results of this optimization for the 1MB cache size where it has been most effective. The optimization does fix the problem with SPECweb99, but it prevents the receive micro-benchmark from reaching its full potential, because our heuristic disables the zero-copy optimization too quickly. We believe that further refinements in our algorithm will lead to a single policy that provides the best performance for each application.

For all of the results shown above, the receive FIFO was 512kB, with 384kB being on-chip due to having 256 VNICs (each VNIC can point to a 1514 byte packet). Figure 12 shows that for the receive micro-benchmark, only 64 VNICs and 96kB of on-chip FIFO are needed. For this benchmark, the receive buffers are not pre-posted, but the low CPU utilization keeps the latency low. Though the number of VNICs necessary to achieve optimal bandwidth will certainly vary from benchmark to benchmark, optimizations to the kernel protocol stack to reduce the amount of queuing should reduce the number required.

7. Conclusion and Future Work

We have described a simple network interface—SINIC—designed to take advantage of integration onto the processor die in support of high-bandwidth TCP/IP networking. SINIC is simpler than conventional NICs in that it avoids the overhead and complexity of DMA descriptor management, instead exposing a raw FIFO interface to the device driver. In spite of its simplicity, detailed full-system simulation results show that SINIC performs as well as a more complex conventional NIC given the same level of integration. These results show that SINIC's simplicity does not incur a performance cost, even when its flexibility is not exploited.

We also presented a novel approach to extending SINIC's FIFO-based interface to allow packet-level parallelism both on transmit and receive. By associating a set of "virtual" FIFO registers with each packet, the V-SINIC interface allows lockless concurrent packet transmission on multiprocessors and enhanced parallelism in receive packet processing. V-SINIC also enables a deferred-copy technique that supports a straightforward implementation of zero-copy receive handling, which we have implemented in the Linux 2.6 kernel. This zero-copy implementation can provide a more than 50% performance improvement on cache-constrained systems.

We believe that simple NICs closely coupled with general-purpose host CPUs, as exemplified by SINIC and V-SINIC, provide far more flexibility and opportunity for optimization than systems in which dedicated processing capability is added to a NIC residing on an I/O bus. In SINIC, the movement of packets into and out of the network FIFOs is controlled directly by the device driver, meaning that these critical operations can be optimized and customized to work with specific operating systems, limited only by the ingenuity of kernel developers. Our Linux zero-copy implementation is a significant optimization but we believe it is not likely to be the only one enabled by SINIC-style interfaces.

Our future work includes evaluation of SINIC and V-SINIC on additional networking benchmarks and further exploration of the SINIC/V-SINIC design space, including sensitivity analysis of the SINIC access latency. Open issues include how best to support encryption and decryption of network traffic and how to virtualize SINIC for virtual-machine systems.

Acknowledgments

This work has been supported by the National Science Foundation under grant CCR-0219640; by gifts from Intel, IBM, and Sun; and by an Intel Fellowship and a Sloan Research Fellowship.

References

[1] Alacritech, Inc. Alacritech / SLIC technology overview. `http://www.alacritech.com/html/tech_review.html`.

[2] Apache Software Foundation. Apache HTTP server. `http://httpd.apache.org`.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, Jul/Aug 2006.

[5] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Performance analysis of system overheads in TCP/IP workloads. In *Proc. 14th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 218–228, Sept. 2005.

[6] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. 2nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 154–165, Feb. 1996.

[7] Broadcom Corp. BCM5706 product brief, 2004. `http://www.broadcom.com/collateral/pb/5706-PB04-R.pdf`.

[8] Broadcom Corporation. BCM1250 product brief, 2003. `http://www.broadcom.com/collateral/pb/1250-PB09-R.pdf`.

[9] J. Chase. *High Performance TCP/IP Networking*, chapter 13, "Software Implementation of TCP". Prentice-Hall, 2003.

[10] J. Corbet. Linux and TCP offload engines. *Linux Weekly News*, Aug. 2005. `http://lwn.net/Articles/148697`.

[11] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *Proc. 14th Ann. Int'l Symp. on Computer Architecture*, pages 189–196, May 1987.

[12] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[13] C. Demerjian. Sun's Niagara falls neatly into multithreaded place. *The Inquirer*, Nov. 2004. `http://www.theinquirer.net/?article=19423`.

[14] W. Feng et al. Optimizing 10-Gigabit Ethernet for networks of workstations, clusters, and grids: A case study. In *Proc. Supercomputing 2003*, Nov. 2003.

[15] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine multicomputer. In *28th Ann. Int'l Symp. on Microarchitecture*, pages 146–156, Dec. 1995.

[16] A. P. Foong, T. R. Huff, H. H. Hum, J. Patwardhan, and G. J. Regnier. TCP performance re-visited. In *Proc. 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, Mar. 2003.

[17] B. Francis. Enterprises pushing 10GigE to edge. *InfoWorld*, Dec. 2004. `http://www.infoworld.com/article/04/12/06/49NNcisco_1.html`.

[18] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server network scalability and TCP offload. In *Proc. 2005 USENIX Technical Conference*, pages 209–222, Apr. 2005.

[19] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proc. 1999 USENIX Technical Conference, Freenix Track*, 1999.

[20] P. Gelsinger, H. G. Geyer, and J. Rattner. Speeding up the network: A system problem, a platform solution. Technology@Intel Magazine, Mar. 2005. `http://www.intel.com/technology/magazine/communications/speeding-network-0305.pdf`.

[21] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proc. Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 111–122, Oct. 1992.

[22] Hewlett-Packard Company. Netperf: A network performance benchmark. `http://www.netperf.org`.

[23] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt. Sampling and stability in TCP/IP workloads. In *Proc. First Annual Workshop on Modeling, Benchmarking, and Simulation*, pages 68–77, June 2005.

[24] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *Proc. 32nd Ann. Int'l Symp. on Computer Architecture*, pages 50–59, June 2005.

[25] Intel Corp. *Intel IXP1200 Network Processor Family - Hardware Reference Manual*, Dec. 2001.

[26] K. Lauritzen, T. Sawicki, T. Stachura, and C. E. Wilson. Intel I/O acceleration technology improves network performance, reliability and efficiently. Technology@Intel magazine, Mar. 2005. `http://www.intel.com/technology/magazine/communications/Intel-IOAT-0305.pdf`.

[27] D. S. Miller. Re: [PATCH] TCP Offload (TOE) - Chelsio. E-mail, Aug. 2005. `http://lwn.net/Articles/148701`.

[28] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[29] S. S. Mukherjee and M. D. Hill. Making network interfaces less peripheral. *IEEE Computer*, 31(10):70–76, Oct. 1998.

[30] T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, June 1968.

[31] National Semiconductor. DP83820 datasheet, Feb. 2001. `http://www.national.com/ds.cgi/DP/DP83820.pdf`.

[32] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19th Ann. Int'l Symp. on Computer Architecture*, pages 156–167, May 1992.

[33] M. Ohmacht et al. Blue Gene/L compute chip: Memory and Ethernet subsystem. *IBM Journal of Research and Development*, 49(2/3):255–264, March/May 2005.

[34] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, Nov. 2004.

[35] A. G. Saidi, N. L. Binkert, L. R. Hsu, and S. K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *Proc. 2005 Workshop on Interaction between Operating System and Computer Architecture (IOSCA)*, pages 33–38, Oct. 2005.

[36] J. Satran, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iscsi. `http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.pdf`, January 2004.

[37] P. Shivam and J. S. Chase. On the elusive benefits of protocol offload. In *NICELI '03: Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 179–184, 2003.

[38] Standard Performance Evaluation Corporation. SPECweb99 benchmark. `http://www.spec.org/web99`.

[39] P. Willmann, H. Kim, S. Rixner, and V. S. Pai. An efficient programmable 10 gigabit Ethernet network interface card. In *Proc. 11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2005.