A Simple Integrated Network Interface for High-Bandwidth Servers

Nathan L. Binkert Ali G. Saidi

Steven K. Reinhardt

{binkertn,saidi,stever}@eecs.umich.edu

Abstract

High-bandwidth TCP/IP networking places a significant burden on end hosts. We argue that this issue should be addressed by integrating simple network interface controllers (NICs) more closely with host CPUs, not by pushing additional computation out to the NICs. We present a simple integrated NIC design (SINIC) that is significantly less complex and more flexible than a conventional DMA-descriptor-based NIC but performs as well or better than the conventional NIC when both are integrated onto the processor die. V-SINIC, an extended version of SINIC, provides virtual per-packet registers, enabling packet-level parallel processing while maintaining a FIFO model. V-SINIC also enables deferring the copy of the packet payload on receive, which we exploit to implement a zero-copy receive optimization in the Linux 2.6 kernel. This optimization improves bandwidth by over 50% on a receive-oriented microbenchmark.

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 [Fra04].

Although users can now plug 10 Gbps Ethernet links into their server systems, getting those systems to keep up with the bandwidth on those links is challenging. One proposed solution is to perform much of the required protocol processing on an intelligent network interface controller (NIC), referred to as a TCP offload engine (TOE). This offload approach reduces host CPU utilization and, perhaps more importantly, reduces the frequency of costly interactions between the CPU and NIC across the relatively high-latency I/O bus. Due to the high transfer rate and the complexity of the TCP protocol, a 10 Gbps TOE requires a substantial amount of general-purpose processing power.

This trend of pushing more intelligence out to the NIC brings to mind Myer and Sutherland's "wheel of reincarnation" [MS68]. 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. Providing computational resources via an additional symmetric CPU imparts more flexibility and generality than using a dedicated, asymmetric, special-purpose processing unit.

In this paper, we propose a system architecture for 10GigE servers that completes the turn of the wheel for NIC design. We strip the NIC down to its most basic components—a pair of FIFOs—supplemented only by a block copy/checksum unit. All other processing is performed by the device driver on a general-purpose host CPU. The driver directly initiates individual transfers between the FIFOs and main memory—a task typically performed by a DMA engine in even the most basic NICs, but which requires non-trivial computational resources at 10 Gbps [WKRP05]. A key factor in enabling this design is the integration of the NIC on the processor die, providing low-latency interaction between the CPU(s) and the NIC control registers.

Using detailed full-system simulation, we show that our Simple Integrated NIC (SINIC) provides performance comparable to a conventional DMA-descriptor-based NIC design when the latter is also integrated onto the processor die, even though SINIC does not provide asynchronous DMA. Both of these designs provide significantly higher performance than a DMA-based off-chip NIC due to their lower CPU access latency and their ability to place incoming DMA data into the on-chip cache [HIT05, BHS+05].

A key potential benefit of exposing the NIC FIFOs to the host CPUs is that kernel software can control the memory destination of each packet explicitly. To illustrate this flexibility, 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 has posted a receive buffer (e.g., by calling `read()`) before the packet's arrival, 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. We remove 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, including lockless parallel access to the FIFOs on chip multiprocessors.

The remainder of the paper begins with a qualitative case for a simple integrated network interface, followed by a description of our SINIC design 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 a Simple Network Interface

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 this integration is not common on high-performance servers today, there are numerous examples in the embedded space (e.g., from Broadcom [Bro03]). Each BlueGene/L node incorporates an integrated 1 Gbps Ethernet NIC for I/O [O+05]. Given available transistor budgets, the potential performance benefits [BHS+05], 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 [Dem04].

For our simple NIC, only the heads of the respective FIFOs and their associated control logic need be integrated on the die. Additional FIFO buffer space and the physical link interface (PHY) can be off-chip. A single processor product with an integrated NIC could thus support multiple physical media (e.g., copper or fiber).

2.2 Simple Versus Conventional NICs

The interface to a conventional Ethernet NIC also consists of a pair of FIFOs plus control and status information. The control and status information is typically exchanged with the CPU through uncached, memory-mapped device registers. A conventional NIC resides on a standard I/O bus (e.g., PCI) that is physically distant from and clocked much more slowly than the CPU, so these uncached accesses may require thousands of CPU cycles to complete [BHS+05]. Providing access to the network data FIFOs via these memory-mapped device registers is impractically slow, so the NIC uses DMA to extend its hardware FIFOs into system memory.

To give the operating system some flexibility in memory allocation, the memory-resident FIFOs are divided into multiple non-contiguous buffers. The address and length of each buffer is recorded in a DMA descriptor data structure, also located in main memory. The transmit and receive FIFOs are represented as lists of DMA descriptors.

To transmit a packet, the device driver creates a DMA descriptor for each of the 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.

The device driver constructs the receive queue by preallocating empty buffers and their associated DMA descriptors. The NIC uses DMA read operations to fetch the descriptors, DMA writes to copy data from its internal receive FIFO into the corresponding buffers, and further DMA writes to mark the descriptors as containing data. In most cases the NIC will also signal an interrupt to the CPU to notify it of the data arrival. The device driver will then process the DMA descriptors to extract the buffer addresses, pass the buffers up to the kernel's protocol stack, and replenish the receive queue with additional empty buffers.

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. [WKRP05] analyzed a commercial 1 Gbps Ethernet NIC that implements this style of DMA queue in firmware, and determined that an equivalent 10 Gbps NIC must sustain 435 MIPS to perform these tasks at line rate, assuming full 1518-byte frames. They proceed to show how this throughput can be provided in a power-efficient way using a dedicated six-way 166 MHz embedded multiprocessor. Note that, other than possibly calculating checksums, this computational effort provides *no* inspection or processing of the packets whatsoever; it merely serves to extend the network FIFOs into system memory where abundant buffering can hide the high latency of CPU/NIC communication. Although these functions could be implemented more efficiently in an ASIC than in firmware, this approach bears the traditional ASIC burdens of cost, complexity, and time to market for no additional functional benefit.

In contrast, a simple NIC that directly exposes the hardware FIFOs to software does not require DMA descriptors at all, avoiding the management overhead. On transmit, the device driver merely copies the packet from the data buffer(s) into the FIFO. As soon as the copy completes, the buffers are free to be reused. On receive, the host CPU is notified via an interrupt if necessary, and the driver copies data from the receive FIFO into a buffer. Our SINIC design, described in detail in Section 3, includes a simple block copy engine to make these copies more efficient, but little else. 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 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.

[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.

2.3 Simple NICs Versus TCP Offload Engines

Rather than integrating the NIC near the CPU, systems can closely couple processing with the NIC by leaving the NIC on an I/O bus and adding 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 [Ala, Bro04].

One disadvantage of the TOE approach is that the associated processing power is dedicated for a single purpose and cannot be reallocated. In contrast, using a general-purpose CPU for protocol processing means that that resource can be used for other purposes when the network is not saturated.

Another 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 not only 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. The fact that the code controlling copies out of the SINIC receive FIFO is part of the device driver, and thus has immediate access to the full kernel code and data structures, 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 [Mog03, RMI$^+$04] and the fact that TOEs provide significant speedups only under a limited set of workload conditions [SC03].

3 The Simple Integrated Network Interface Controller (SINIC)

This section describes the detailed design of one possible low-level NIC interface—our simple integrated NIC (SINIC)—and its associated device driver. 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 extend SINIC to enable this parallelism by providing virtual per-packet registers, a design we call V-SINIC. Finally we describe how we used 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.

The core of the SINIC interface consists of four memory-mapped registers: RxData, RxDone, TxData, and TxDone. 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 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 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.

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.

SINIC implements a single copy engine per FIFO.[2] As a result, the CPU must wait for the previous copy to complete before initiating another copy. Because individual buffer transfers are relatively fast, the driver simply busy waits when it needs to perform multiple copies. SINIC enables more efficient synchronization

[2]These engines share a single cache port, so only one can perform a transfer in any given cycle.

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 semantics (e.g., Linux `sk_buffs`) into the NIC's DMA descriptor format. With SINIC, there are no descriptors. When a packet must be transmitted, the device driver simply loops over each portion of the packet buffer initiating a transmit with a programmed I/O (PIO) write to TxData, busy waiting 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. 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. This overhead can be avoided on the receive side by copying data directly from the NIC FIFO into the user buffer. Unfortunately, this "zero-copy" behavior is practically impossible to achieve with a conventional DMA-descriptor-based NIC, as receive packet buffers must be posted to the NIC before the driver has any idea for which connections the arriving packets will be destined.

V-SINIC's deferred-copy capability enables a straightforward implementation of zero-copy receives in the Linux 2.6 kernel. We enhanced Linux's `sk_buff` structure to be able to indicate that the referenced data was resident in the V-SINIC FIFO (including the appropriate VNIC index). We also modified the `skb_copy_bits()` and `skb_copy_datagram_iovec()` kernel functions—which copy the contents of an `sk_buff` to a kernel or user buffer, respectively—to recognize this encoding and request the copy directly from the FIFO via the VNIC.

4 Related Work

On-chip integrated network interfaces have appeared before in the context of fine-grain massively parallel processors. Henry and Joerg [HJ92] 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 [DCC+87], M-Machine [FKD+95], and *T [NPA92] research projects, and IBM's BlueGene/L [O+05]. Mukherjee and Hill [MH98] 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. 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 what the kernel would like to see for TCP/IP processing.

In the TCP/IP domain, a few other groups have investigated alternatives to offloading. Binkert et al. [BHS+05] 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 [LSSW05] that explicitly discounts TOEs in favor of a "platform solution" [GGR05]. Intel researchers proposed a "TCP onloading" model in which one CPU of an SMP is dedicated to TCP processing [RMI+04]. Our SINIC model is complementary to this approach: the dedicated CPU would likely benefit . Another Intel paper describes "direct cache access" I/O [HIT05], 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 [Cha02]. 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. Unfortunately, this technique is challenging to apply as it requires packets to be as large as physical pages and suitably aligned. 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 sophisticated NIC support, RDMA is a significant change to both applications and protocols, and requires support on both ends of a connection. Our V-SINIC approach is most closely related to that of Afterburner [DWB+93], an experimental NIC that combined significant on-board 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. The amount of buffer space required is proportional to the product of the network bandwidth and the CPU/NIC latency, and quickly becomes significant. Although Afterburner was relatively closely coupled to the CPU (plugging into a graphics card slot on an HP workstation), 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.[3]

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

Much of the execution time spent by an network intensive benchmark is not running application code, but rather kernel code such as device drivers and the TCP/IP stack. This distribution prevents the use of a simulator that functionally emulates syscalls from providing meaningful results for a network benchmark. To mitigate this issue we turn to a full-system simulator called M5 [BHR03]. This simulator models the Alpha

[3]Our simulated implementation has up to 380 KB of space in the receive FIFO—256 VNICs times 1514 bytes per packet—but we believe that our performance will not suffer with a smaller FIFO. We intend to experiment with this parameter in the near future.

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	8MB, 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

Tsunami platform with enough fidelity to boot an unmodified Linux 2.6 kernel and run Alpha PALcode. Additionally it has been validated against a real Alpha XP1000 (a member of the Tsunami family) [SBHR05].

The memory and I/O system in a network simulation are key to the performance the system can achieve. M5 has a simple bus model that can be used to model interconnect of configurable width and speed. Additionally a bus bridge exists that can join any two busses together, even ones of different speed. With these simple components a I/O system can be created that can model a device hanging off an I/O bridge, such as a commodity network controller today.

For our experiments that use a commodity NIC M5 models a National Semiconductor DP83820 [Nat01] Gigabit Ethernet device. The model is of a high enough fidelity to support the standard Linux driver for this device, however a bug is fixed both in the device model and driver that allows the NIC to DMA to unaligned addresses. Because of the high interrupts rate possible at 10Gbps a fixed delay scheme is used in the NIC to bound the rate of interrupts that the CPU can generate to one every 10us. This Ethernet device is connected to another Ethernet device using an lossless Ethernet link that has 10Gbps of bandwidth.

Table 1 lists the other parameters we used for our simulation. Since the memory system we are modeling is similar to that of a Opteron system, we configured the latency memory, bus bridges and peripheral devices to match numbers measured on an AMD Opteron server.

5.2 Benchmarks

All of the benchmarks used to evaluate the performance of our SINIC design were simulated in a client-server configuration where the system under test, either client or server depending on the benchmark, was modeled in detail and the stresser was modeled functionally with a perfect memory system so as to not be the bottleneck.

Netperf [Hew] is a network microbenchmark suite developed at Hewlett-Packard. It contains a variety of microbenchmarks for testing the bandwidth characteristics of sockets on top of TCP or UDP. Out of the available tests, we use the TCP stream benchmark, a transmit benchmark as well as the TCP maerts benchmark, a receive benchmark. In both these cases after setting up a socket one machine generates data as fast as possible by calling `send()`. Normally this call returns as soon as the data is copied out of the userspace buffer, however if the socket buffer is full the call will block until space is available. In this way the benchmark is self-tuning. Similarly the second machine attempts to sink data as fast as possible by calling `receive()` as fast as possible. 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.

In addition to the standard stream and maerts benchmark we created two variants that each use more than one connection, writing and reading data from them in a round-robin fashion. These benchmarks which we term stream-multi and maerts-multi provide more streams for the kernel and NIC to manage while guaranteeing that they are all of similar bandwidth. With several independent streams this isn't the case, because one stream could end up getting the majority of the available bandwidth making it harder to draw conclusions from the this microbenchmark.

SPEC WEB99 [Sta] is a popular benchmark that is used for evaluating the performance of webservers. 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 [Apa] with the mod_specweb99 CGI scripts. These scripts replaced the reference implementation, with a more optimized implementation written in C and are frequently used in the results on the SPEC website.

The standard SPEC WEB99 client isn't well suited for a simulation environment. A standard SPEC WEB99 score is based on the maximum number of simultaneous clients that the server can support while meeting some minimum bandwidth and response time guarantees. Each client requests a small amount

Benchmark	Warm-up Time	Sampling Time
Netperf Single-stream	100M	50M
Netperf Multi-stream	500M	100M
SPECWEB99	1B	400M
iSCSI	1B	100M

Table 2: Simulated System Parameters

of data and thus a SPEC WEB99 score is normally obtained by a large testbed of clients and an interactive tuning process find the maximum number of clients for a particular server configuration. This approach isn't practical for our tests because of the large slowdown simulation incurs. For our purposes we aren't concerned with the SPEC WEB99 score attainable by the machine under test, but rather are simply interested in the performance characteristics of a web server workload. To this end we chose to use a different client based on the Surge traffic generator [BC98] that preserves the same statistical distribution as the original client, but is able to scale it's performance up to a point that it can saturate the server.

iSCSI [SSCZ04] is a new standard to use the SCSI protocol on top of a TCP/IP connection allowing a initiator (client) to access a target (server) in a similar manner as it would locally with a SCSI host adapter connected to a SCSI device. Because of its use of TCP/IP as a connection layer protocol and Ethernet as a link layer protocol it is 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. The target was configured to not have a real I/O backing store, and instead just return data immediately. This simplification is reasonably in our test since we are not concerned with disk I/O performance. On top of the iSCSI client we run a custom benchmark that uses Linux's asynchronous I/O(AIO) facilities to continuously have multiple outstanding reads to the iSCSI disk in-flight at once. As soon as a read completes, a new location to read is selected and it is reissued to the disk. We benchmark both the target and the initiator. In each case the better system can provide responses faster or with less overhead allowing more requests to be sent and thus increasing the bandwidth seen on a link between client and server.

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 changing it is reasonable in a controlled environment, it won't be used for commodity traffic on the Internet and thus we fix it in our experiments.

Running the above workloads to completion in simulation is infeasible due to the massive slowdown encured. 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 are at a programmatically good point to checkpoint. At this point the simulator checkpoints all program state and can then restore this state at a later time, warm-up the caches and TLB, and switch to a detailed model to gather results. For our experiments we warmed-up and then simulated 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 change from simple to detailed simulation [HSBR05].

6 Results

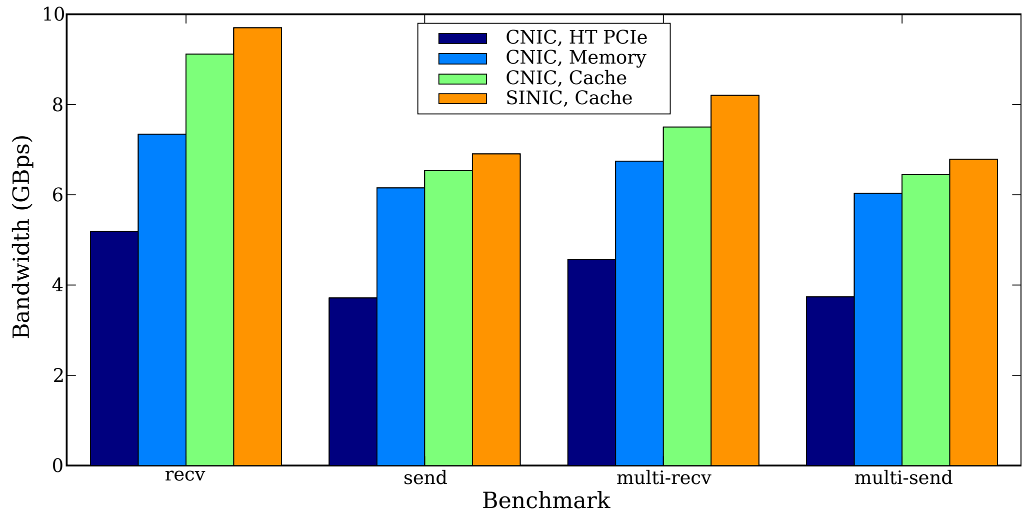

Figure 1: Micro-Benchmark Bandwidth - Measurements of achieved bandwidth of the various micro-benchmarks with CPUs running at 4GHz.

In this section, SINIC with direct attachment to the on-chip last level cache is compared to a conventional NIC with various points of attachment. Additionally, we explore the effectiveness of the zero-copy optimization. To set a baseline, measurements are taken with a conventional NIC attached to a PCI Express-like bus, which is connected to the CPU by way of an I/O bridge chip using a HyperTransport-like interconnect. The on-chip configurations of a conventional NIC include attachment on the far-side of the last level of on-chip cache, giving it DMA access to the on-chip memory controller, but not the cache and attachment to the near-side of the last level of on-chip cache, giving it access to the last level cache. SINIC is always attached to the near-side of the last level cache.

Figure 1 shows the performance of these configurations when running the Netperf microbenchmarks. The Commodity NIC (CNIC) attachments clearly show that tighter integration of the NIC in the system increases performance showing a better than 60% improvement in all cases. There are two main reasons for this. First, accessing the device has a much lower latency and the device can place data directly into the last level of cache, thus reducing the amount of memory traffic and cache misses. Second, placing data directly in the cache can reduce the number of cache misses per kilobyte of data transmitted from one per block to zero [BHS+05, HIT05].

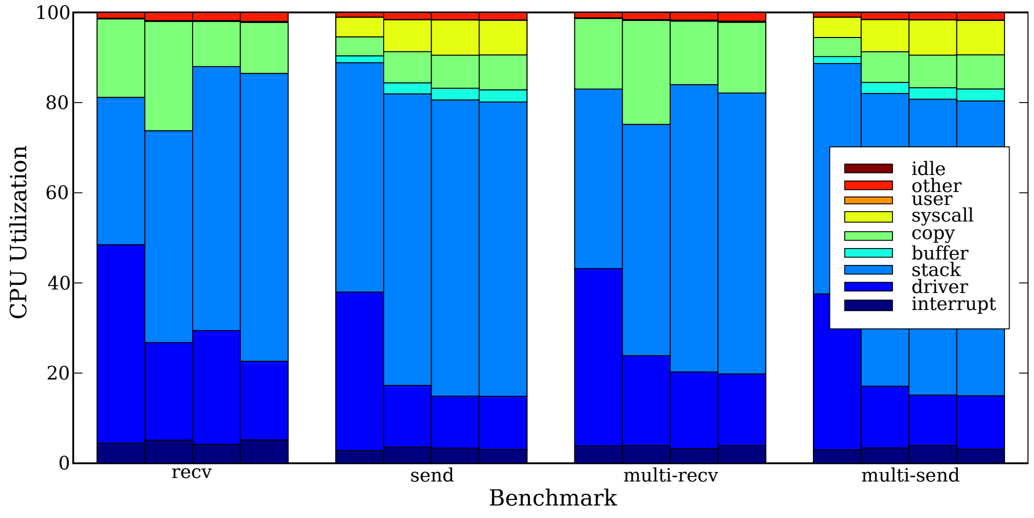

Figure 2: Micro-Benchmark CPU Utilization Breakdown - Breakdown of CPU utilization for the various micro-benchmarks with CPUs running at 4GHz.

In Figure 1 we also see that SINIC, which is attached on the near side of the cache, outperforms the more complex CNIC attached at the same position. As mentioned above, CNIC designers implement a DMA descriptor mechanism for managing copies to and from the network. SINIC doesn't have this overhead even though the data copies must be initiated with programmed I/O. The device's close attachment to the CPU makes the latency tolerance that the DMA descriptors provide unnecessary.

In Figure 2 we show a breakdown of where the CPU spent its time. Note the reduction in driver time seen when comparing the PCI Express attached NIC and the other attachments. Because the device is closer, accessing its device registers is a far cheaper operation, providing more CPU time for processing the data.

When considering more complex macro-workloads, which are often more compute bound, one may worry that pushing the NIC state machine into the device driver would incur an unacceptable overhead

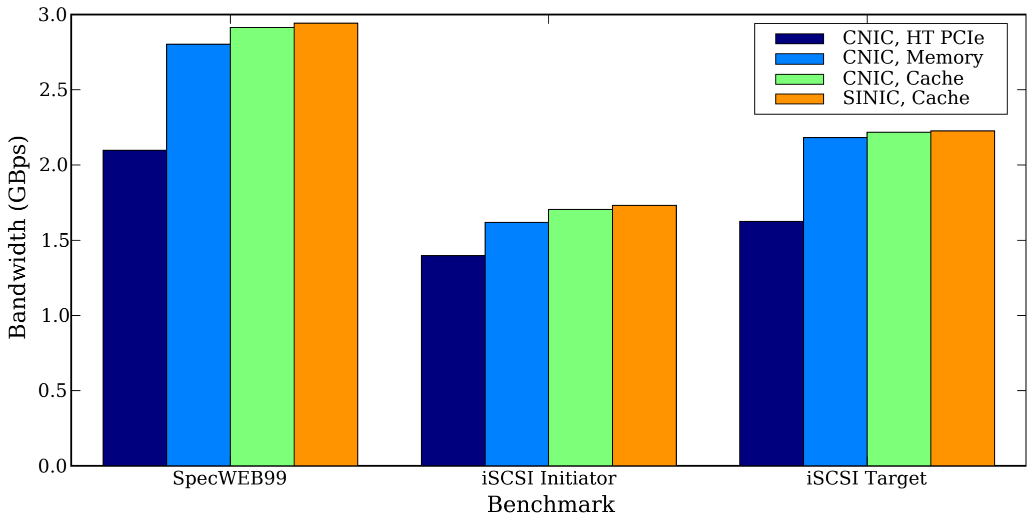

Figure 3: Macro-Benchmark Bandwidth - Measurements of achieved bandwidth of the various macro-benchmarks with CPUs running at 4GHz.

when compared to a DMA engine doing the work. Figure 3 shows that this is not the case. The increased computation due to managing the state machine is offset by the removal of the the descriptor management code from the device driver because these two functions are actually similar. Thus, even for more complex workloads, including CPU bound ones like SPECWeb99, we do not observe a decrease in performance.

We show the overall performance results of the zero-copy optimization in Figure 4. Repeating the result shown in Figure 1, SINIC is able to saturate the network with a large (4-8 MB) L2 cache, because its 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 to additional overheads in the device driver.[4] 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. Figure 5 shows that the number of cache misses is strongly correlated to 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 because network data is read directly from

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

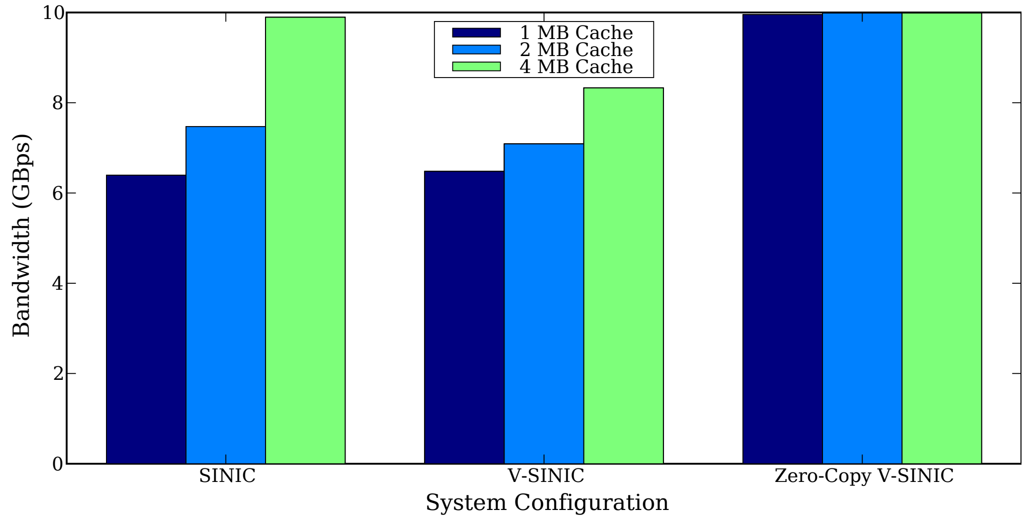

Figure 4: Zero-Copy Bandwidth - Achieved bandwidth for the receive microbenchmark with CPUs running at 4GHz.

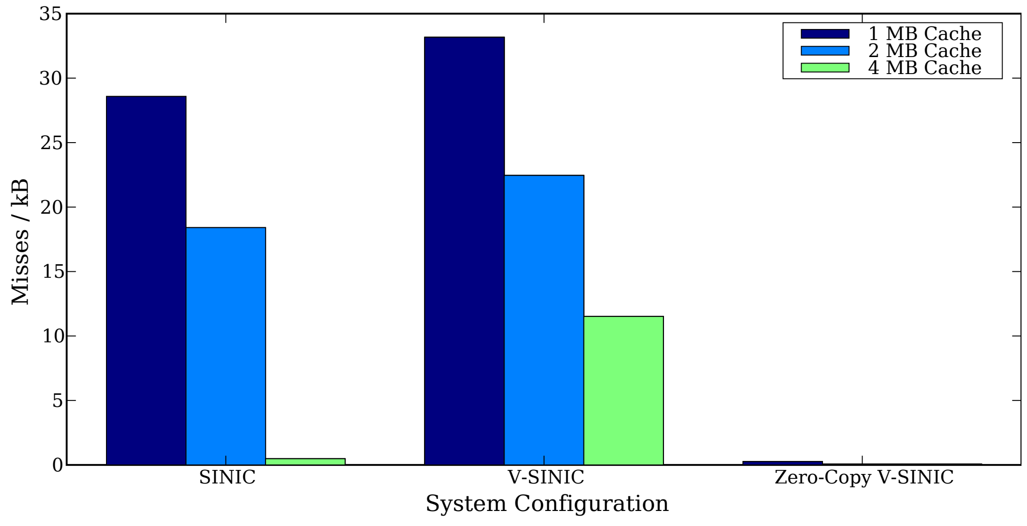

Figure 5: Zero-Copy Cache Miss Rates - Cache miss rate in misses per kilobyte of data transferred for the receive microbenchmark with CPUs running at 4GHz.

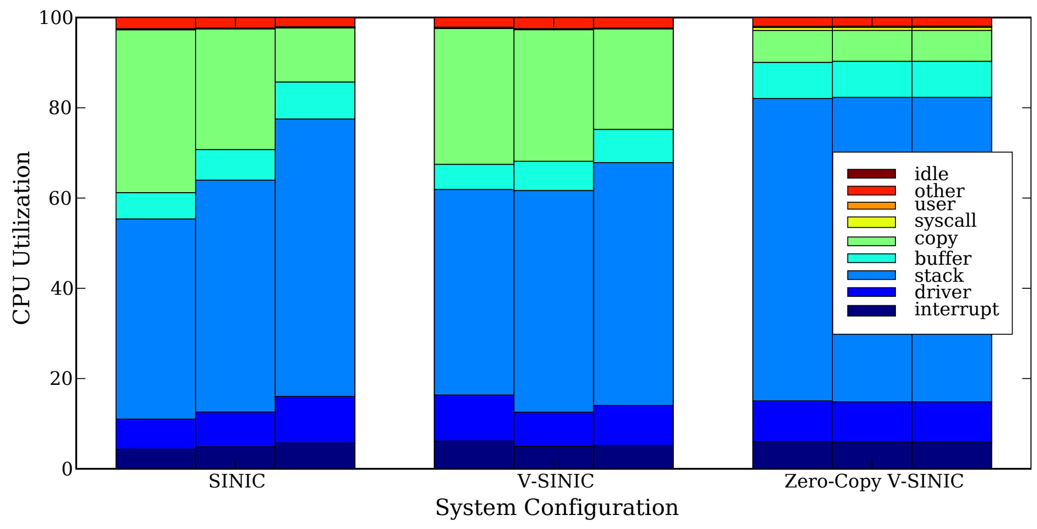

Figure 6: Zero-Copy CPU Utilization Breakdown - Breakdown of CPU utilization for the receive microbenchmark with CPUs running at 4GHz.

the FIFO rather than from the memory system. Figure 6 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 on the V-SINIC copy engine to copy data from the FIFO to the user buffer.

7 Conclusion and Future Work

We have described a simple network interface—SINIC—designed to be integrated onto the processor die to support 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 or better than a conventional NIC given the same level of integration.

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 to 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 is shipped out 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.

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 and number of VNICs available for V-SINIC. Open issues include how best to support encryption and decryption of network traffic and how to virtualize SINIC for virtual-machine systems.

References

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

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

[BC98] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[BHR03] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, February 2003.

[BHS$^+$05] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven 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 210–228, September 2005.

[Bro03] Broadcom Corporation. BCM1250 product brief, 2003. http://www.broadcom.com/collateral/pb/1250-PB09-R.pdf.

[Bro04] Broadcom Corp. BCM5706 product brief, 2004. http://www.broadcom.com/collateral/pb/5706-PB04-R.pdf.

[Cha02] Jeff Chase. *High Performance TCP/IP Networking*, chapter TCP Implementation. Prentice-Hall, February 2002.

[DCC+87] 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.

[Dem04] Charlie Demerjian. Sun's Niagara falls neatly into multithreaded place. *The Inquirer*, November 2004. http://www.theinquirer.net/?article=19423.

[DWB+93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[FKD+95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *28th Ann. Int'l Symp. on Microarchitecture*, pages 146–156, December 1995.

[Fra04] Bob Francis. Enterprises pushing 10GigE to edge. *InfoWorld*, December 2004. http://www.infoworld.com/article/04/12/06/49NNcisco_1.html.

[GGR05] Pat Gelsinger, Hans G. Geyer, and Justin Rattner. Speeding up the network: A system problem, a platform solution. Technology@Intel Magazine, March 2005. http://www.intel.com/technology/magazine/communications/speeding-network-0305.pdf.

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

[HIT05] Ram Huggahalli, Ravi Iyer, and Scott 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.

[HJ92] Dana S. Henry and Christopher 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, October 1992.

[HSBR05] Lisa R. Hsu, Ali G. Saidi, Nathan L. Binkert, and Steven K. Reinhardt. Sampling and stability in TCP/IP workloads. In *Proc. First Annual Workshop on Modeling, Benchmarking, and Simulation*, pages 68–77, June 2005.

[LSSW05] Keith Lauritzen, Thom Sawicki, Tom Stachura, and Carl E. Wilson. Intel I/O acceleration technology improves network performance, reliability and efficiently. Technology@Intel magazine, March 2005. http://www.intel.com/technology/magazine/communications/Intel-IOAT-0305.pdf.

[MH98] Shubhendu S. Mukherjee and Mark D. Hill. Making network interfaces less peripheral. *IEEE Computer*, 31(10):70–76, October 1998.

[Mog03] Jeffery C. Mogul. TCP offload is a dumb idea whose time has come. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[MS68] T. H. Myer and I. E. Sutherland. On the design of display processors. *Communications of the ACM*, 11(6):410–414, June 1968.

[Nat01] National Semiconductor. DP83820 datasheet, February 2001. `http://www.national.com/ds.cgi/DP/DP83820.pdf`.

[NPA92] 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.

[O$^+$05] 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.

[RMI$^+$04] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, November 2004.

[SBHR05] Ali G. Saidi, Nathan L. Binkert, Lisa R. Hsu, and Steven 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)*, October 2005.

[SC03] Piyush Shivam and Jeffrey 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.

[SSCZ04] Julian Satran, Costa Sapuntzakis, Millikarjun Chadalapaka, and Efri Zeidner. iscsi, January 2004. `http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.pdf`.

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

[WKRP05] Paul Willmann, Hyong-youb Kim, Scott Rixner, and Vijay S. Pai. An efficient programmable 10 gigabit Ethernet network interface card. In *Proc. 11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, February 2005.