The Performance Potential of an Integrated Network Interface

Nathan L. Binkert, Ronald G. Dreslinski, Erik G. Hallnor, Lisa R. Hsu,
Steven E. Raasch, Andrew L. Schultz, and Steven K. Reinhardt

Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{binkertn, rdreslin, ehallnor, hsul, sraasch, alschult, stever}@eecs.umich.edu

Abstract

High-bandwidth TCP/IP networking is a core component of current and future computer systems. Though networking is central to computing today, the vast majority of end-host networking research focuses on the current paradigm of the network interface being merely a peripheral device. Most optimizations focus solely on software changes or on moving some of the computation from the primary CPU to the off-chip network interface controller (NIC). We present an alternative approach for achieving high performance networking. Rather than increasing the complexity of the NIC, we directly integrate a conventional NIC on the CPU die.

To evaluate this approach, we have developed a simulation environment specifically targeted for networked systems. It simulates server and client systems along with a network in a single process. Full-system simulation captures the execution of both application and OS code. Our model includes a detailed out-of-order CPU, event-driven memory hierarchy, and Ethernet interface device. Using this simulator, we find that tighter integration of the network interface can provide benefits in TCP/IP throughput and latency. We also see that the interaction of the NIC with the on-chip memory hierarchy has a greater impact on performance than the raw improvements in bandwidth and latency that come from integration.

This material is based upon work supported by the National Science Foundation under Grant No. 0219640. This work was also supported by gifts from Intel and IBM, an Intel Fellowship, a Lucent Fellowship, and a Sloan Research Fellowship.

1. Introduction

In the past decade, the role of computers in society has undergone a dramatic shift from standalone processing devices to multimedia communication portals. As a result, TCP/IP networking has moved from an optional add-on feature to a core system function. Ubiquitous TCP/IP connectivity has also made network usage models more complex and varied: general-purpose systems are often called upon to serve as firewalls, routers, or VPN endpoints, while IP-based storage networking is emerging for high-end servers. At the same time, available end-system network bandwidths have increased faster than Moore's Law: from 1995 to 2002, the IEEE Ethernet standard evolved from a top speed of 100 Mb/s to 10 Gb/s, a hundred-fold improvement, while in the same period, the 18-month doubling rate of Moore's Law indicates a mere 25x increase. Combining these trends, it is apparent that TCP/IP network I/O can no longer be considered an afterthought in computer system design.

Nevertheless, for historical reasons, network interface controllers (NICs) in mainstream computers continue to be treated as generic peripheral devices, connected through standardized I/O buses. Because I/O standards evolve relatively slowly, their performance lags behind that of both CPUs and networks. As a result, it is currently possible to purchase a full-duplex 10 Gbps Ethernet card capable of 20 Gbps total bidirectional network throughput that communicates at best over a 133 MHz 64-bit PCI-X bus with a theoretical peak throughput of 8.5 Gbps.[1] Much like the bottleneck of the broader Internet is often in the "last mile" connecting to consumers' homes, the bottleneck for a high-speed LAN is often in the last few inches from the NIC to the CPU/memory complex. Higher I/O bandwidths

[1] http://www.intel.com/network/connectivity/products/pro10GbE_LR_server_adapter.htm

based on the PCI-X 2.0 and PCI Express standards are emerging, but the general trend remains. (As somewhat of a stopgap measure, Intel has introduced a feature called the "Communication Streaming Architecture" (CSA) [12] on some of its chipsets, which is basically a dedicated I/O bus solely for the NIC's use.) Furthermore, these future standards do not fundamentally reduce the latency of communication between the CPU and the network interface, which currently stands at thousands of CPU cycles (see Section 3.4).

One approach to addressing this bottleneck is to optimize the interface between the NIC and the CPU [4, 5, 7, 8, 23]. Most proposals in this area focus on redesigning the hardware interface to reduce or avoid overheads on the CPU, such as user/kernel context switches, memory buffer copies, segmentation, reassembly, and checksum computations. The most aggressive designs, called TCP offload engines (TOEs), attempt to move substantial portions of the TCP protocol stack onto the NIC [2]. These schemes address the I/O bus bottleneck by having the CPU interact with the NIC at a higher semantic level and thus less frequently. More recently, some researchers have addressed I/O bus bandwidth directly, eliminating bus transfers by using NIC memory as a cache [24, 13].

Unfortunately, modifications to the CPU/NIC interface require specialized software support on the CPU side. The task of interfacing the operating system to a specific device such as a NIC falls to a device driver. However, in a typical OS, the network protocol stacks are encapsulated inside the kernel, and the device driver is invoked only to transfer raw packets to and from the network. Even simple interface optimizations generally require protocol stack modifications outside the scope of the device driver, meaning that hardware companies must wait for OS vendors to accept, integrate, and deploy these changes before the optimization can take effect. For example, hardware checksum support is useless unless the kernel stack has the capability of detecting this feature and disabling its own software checksum—a feature that is common today, but was not just a few years ago. (In fact, this feature is missing in the HP/Compaq Tru64 5.1 release from September 2000 that we use on our simulated systems.) Shifting significant amounts of protocol processing to the NIC requires even more radical rewiring of the kernel, which complicates and may even preclude deployment of future protocol optimizations.

This paper examines a straightforward alternative to sophisticated NIC interfaces: direct integration of a conventional NIC on the CPU die. Unlike adding intelligence to the NIC, this approach does not require significant software changes; in fact, existing device drivers can be re-used with at most minor modifications. Although an integrated NIC provides less flexibility than an add-in card, the dominance of Ethernet makes the choice of link-layer protocol a non-issue. A single 10 Gb/s connection (perhaps 40 or 100 Gb/s in the future) should be more than adequate for a significant fraction of the market, and could be split with an off-chip switch into multiple slower connections as necessary. Single-chip multiprocessor server devices could sport a small number of NICs as a product differentiator. High-end multi-chip systems could share integrated NICs easily across their inter-chip interconnect, much as modern systems such as those based on AMD Opteron processors exploit their HyperTransport channels to share DRAM and I/O devices [1].

The end goal of this paper is not to suggest that integration will be a panacea; I/O bus latency and bandwidth are far from the only inefficiencies in conventional TCP/IP protocol processing. An integrated NIC does not directly address overheads such as data copying and interrupt processing that primarily involve the CPU and memory system rather than the I/O bus. However, we feel that current solutions such as TCP offload engines are to some extent products of their environment: moving protocol processing to a loosely coupled I/O adapter may make sense when a loosely coupled I/O adapter is the only easily upgraded component, but longer-term system architecture research should encompass a broader range of possibilities. Thus a conventional but tightly integrated NIC may be a better starting point for considering further interface optimization than the status quo, and may well lead to a very different set of optimizations. We examine one option for integrated NICs: whether the NIC sits in front of or behind the last-level on-chip cache. Further studies are left for future work.

Evaluation is a key challenge in investigating alternative network system architectures. Analyzing the behavior of network-intensive workloads is not a simple task, as it depends not only on the performance of several major system components—processors, memory hierarchy, network adapters, etc.—but also on the complex interactions among these components. As a result, researchers proposing novel NIC features generally prototype them in hardware [7] or emulate them using programmable NICs [4, 5, 8, 23, 13]. Unfortunately, this approach is not feasible for our proposal, which involves modifying not the NIC itself but its location in the system—and modifying the processor die at that. We have therefore developed a simulation environment specifically targeted for networked systems. It simulates server and client systems along with a simple network in a single process. Full-system simulation captures the execution of both application and OS code. The server model includes a detailed out-of-

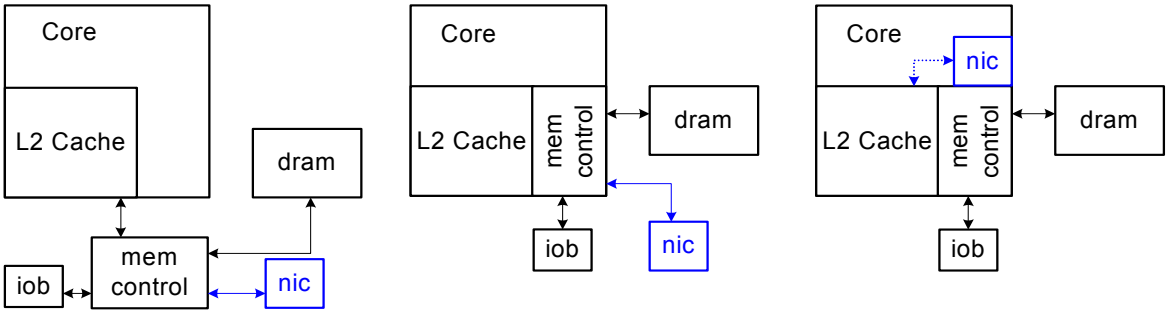

Figure 1: NIC placement options within the system architecture. (a) CSA; (b) HTX; (c) OCM/OCC.

order CPU, an event-driven memory hierarchy, and a detailed Ethernet interface device.

Using our simulator, we model the effects of NIC integration on the performance of several TCP throughput and latency microbenchmarks. We find that an on-die NIC provides higher bandwidth and lower latency than even an aggressive future off-chip implementation. We also see that the interaction of the NIC with the on-chip memory hierarchy has a greater impact on performance than the raw improvements in bandwidth and latency that come from integration.

The remainder of the paper begins with a discussion of the options for NIC placement that we compare in this paper. We describe our simulation environment in Section 3 and our benchmarks in Section 4. Section 5 presents preliminary simulation results. Section 6 discusses related work, and Section 7 presents our conclusions and future work.

2. NIC Placement Options

Figure 1 illustrates the four main configuration options that we evaluate in this paper. The first two represent off-chip NIC architectures and the third an on-chip NIC.

In the first configuration, the memory and the NIC both attach to the memory controller hub. The NIC itself gets a dedicated 64-bit 266 MHz port (2 GB/s) on the memory controller hub. This configuration is analogous to Intel's Communication Streaming Architecture [12] and will be referred to as CSA.

The NIC and memory are both connected directly to the CPU chip in the second configuration. The NIC is connected via a dedicated 6.4 GB/s channel analogous to HyperTransport. We refer to this configuration as HTX.

The final configuration represents the focus of this work, integration of the NIC on the CPU die. We derive two experimental configurations that differ in how they connect to the on-chip memory hierarchy. The first of these configurations attaches the NIC directly to the on-chip memory bus, between the L2 cache (the last level of on-chip cache) and the memory controller. We refer to this configuration as OCM. In this configuration, all NIC memory writes (e.g., for received data) go through the on-chip memory controller to DRAM. As in all other configurations, NIC memory reads (for transmitted data) are directed at DRAM, but will be sourced by the cache hierarchy if the data exists there in the modified state. We assume that this L2/memory controller connection is engineered merely to be adequate for the bandwidth it connects. In our case, 4 bytes per CPU cycle at 4 GHz is adequate to roughly cover two HyperTransport channels and the DRAM interface, all three at 6.4 GB/s each.

The second on-chip NIC configuration attaches the NIC to the bus between the L1 and L2 caches. In this configuration, which we call OCC, the NIC accesses memory through the L2 cache. OCC has two advantages over OCM. First, NIC accesses that hit in the L2 can take advantage of the much higher bandwidth of the L1/L2 interconnect, which we assume is sized to deliver a 64-byte cache line to the L1 in a single CPU cycle. Second, data written by the NIC (data received from the network) is placed into the L2 cache, so the CPU need not go off chip if it accesses that data before it is displaced out of the cache. Other researchers have reported substantial benefits from placing network data directly in the cache hierarchy [17], though no detailed analysis was provided.

3. Simulation Platform

As discussed in the introduction, evaluation of NIC architecture alternatives is usually done by emulation on a programmable NIC or by hardware prototyping. Unfortunately, while these approaches allow modeling of different NICs in a fixed system architecture, they do not lend themselves to modeling a range of system architectures as we have described in the previous section. We thus turn to simulation for our investigation.

Network-oriented system architecture research requires a simulator capable of full-system modeling, enabling it to run OS and application code, as well as a reasonably detailed timing model of the I/O and networking subsystem. The following sections (3.1-3.3) describe each of these aspects of our simulator in turn. Section 3.4 discusses the system parameters we use in this paper.

3.1 Full System Simulation

The bulk of network processing activity occurs in the OS kernel. Thus conventional architectural simulators, which execute only user-level application code with functional emulation of kernel interactions, would not provide meaningful results for networking workloads.

While a few full-system simulators exist [14, 15, 21, 22], none provided the detailed network I/O modeling we required, and none seemed easy to modify to provide this feature. We thus decided to extend our existing application-only architectural simulator, which executes the Alpha ISA, to support full-system simulation. This process included implementing true virtual/physical address translation, processor- and platform-specific control registers (enabling the model to execute Alpha PAL code), and disk, timer, and other peripherals. Because we used SimOS/Alpha [21] as a reference platform for development, we functionally emulate the same Alpha 21164 processor and DEC Turbolaser platform as that system. Although the OS and PAL code use 21164-specific processor control registers, our performance model (described in Section 3.4) more closely matches a 21264, and also supports all user-visible 21264-specific Alpha ISA extensions. We also updated our platform model relative to SimOS/Alpha so we could boot a much more recent version of HP's Tru64 commercial Unix operating system (5.1 vs. 4.0).

We also enhanced our detailed CPU timing model to capture the primary timing impact of system-level interactions. For example, we execute the actual PAL code flow for handling TLB misses; for a memory barrier instruction, we flush the pipeline and stall until outstanding accesses have completed; write barriers prevent reordering in the store buffer; and uncached memory accesses (e.g. for programmed I/O) are performed only when the instruction reaches the commit stage and is known to be on the correct path.

To provide deterministic, repeatable simulation of network workloads, as well as accurate simulation of network protocol behavior, our simulator models multiple systems and the network interconnecting them in a single process. Implementing this capability was simplified by the object-oriented design of our simulator: creating another system requires simply instantiating another set of objects modeling another CPU, memory, disk, etc.

3.2 Memory and I/O System Model

The memory and I/O systems are a key determinant of networking performance. We use a handful of simple component models to construct system models representing those of Section 2.

We use a single bus model of configurable width and clock speed to emulate all of the interconnects in the system. This model provides a split-transaction protocol, and supports bus snooping for coherence. Transaction sizes are variable up to a maximum of 64 bytes. Our DRAM, NIC, and disk controller models incorporate a single slave interface to this bus model. Our cache model includes a slave interface on the side closer to the CPU and a master interface on the side further from the CPU. Note that this model is optimistic for bidirectional point-to-point interconnects such as PCI Express and HyperTransport, as it assumes that the full bidirectional bandwidth can be exploited instantaneously in either direction.

We also have a bus bridge model that interfaces two busses of potentially different bandwidths, forwarding transactions in both directions using store-and-forward timing. This model is used for the I/O bridges (labeled "iob") shown in Figure 1 and for the memory controller (which in our model simply bridges between e.g. the front-side bus and the controller/DRAM bus—DRAM timing is modeled in the DRAM object itself).

3.3 Ethernet Device Model

Our Ethernet NIC model does not mimic any specific real-world device, but it closely resembles the design of typical Gigabit Ethernet adapters found on the market today. The model focuses its detail on the main packet data path and the system-side I/O interface. Three logical blocks comprise the device model: the device itself, the physical interface, and the link. A block diagram of the design of the overall model is given in Figure 2.

The MAC portion of the model manages device programming registers, DMA to and from the device, and the assembling and buffering of packet data. The device model fully participates in the memory system timing, interfacing to the bus model described in Section 3.2 for both DMA transactions and programmed I/O requests to device control registers. DMA transactions are fully coherent with the cache hierarchy.

The physical interface model is a functional component that moves data from the transmit buffer to the link or passes data from the link back to the transmit buffer. Since there is no buffering in the physical inter-

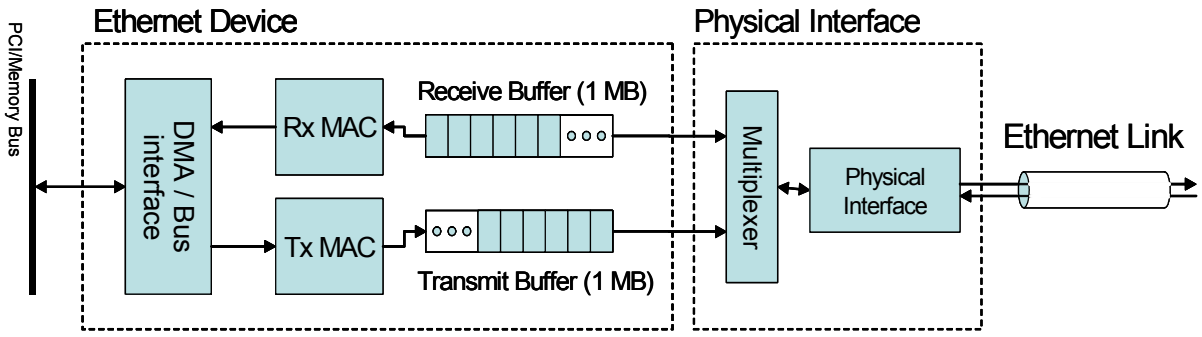

Figure 2: Block diagram of the Ethernet device model

face and it represents negligible delay, its timing impact is not modelled.

The Ethernet link models a lossless, full-duplex link of configurable bandwidth. The latency of a packet traversing the link is simply determined by dividing the packet size by that bandwidth. Since we are essentially modelling a simple wire, only one packet is allowed to be transmitted in each direction at any given time. The sender and receiver keep track of when the link is busy and are responsible for waiting until the link is free and picking up data when it arrives. If the sender attempts to send a packet to a busy link, or the receiver does not have the capacity to buffer a packet incoming packet from the link, the packet is dropped.

In the process of running our workloads, we discovered that we had to add interrupt coalescing to our model in order to achieve high bandwidths. Simple NICs interrupt the CPU every time a packet is transmitted or received. Unfortunately, at high bandwidths, the resulting interrupt rate becomes unbearable. For example, a 10Gbps link with minimum frame size (54 bytes) would cause a whopping 21M interrupts per second. The overhead of handling that many interrupts swamps the CPU, leaving it unable to actually process packets at the desired rate. Interrupt coalescing is a standard term for various techniques to reduce the number of interrupts delivered to the CPU. We use a fixed-delay scheme, in which the device uses a timer to defer delivering a packet interrupt for a specified time (in most of our experiments, we use 20 ms). Once the timer is running, it is not reset when additional packets are sent or received, so a single interrupt will service all packets that are processed during the timer interval. This technique puts an upper bound on the device's interrupt rate at the cost of some additional latency under light loads.

3.4 Platform Parameters

The parameters we used in modeling the configurations of Section 2 are listed in Table 1. We are trying to model the approximate characteristics of these systems, rather than absolute performance, so some of these parameters are approximations. In addition to the parameters below, it is worth noting that we also add a 10 ns latency penalty for each bus bridge that connects devices on separate chips.

Table 1: System Parameters

Parameter	Value
Frequency	4 Ghz or 10 Ghz
Front-end Pipeline	10 cycles fetch-to-decode 5 cycles decode-to-dispatch
Fetch Bandwidth	Up to 4 instructions per cycle
Branch Predictor	Hybrid local/global (ala 21264) Global: 13-bit history, 8K PHT Local: 2K 11-bit history regs, 2K PHT Choice: 13-bit global hist, 8K PHT
BTB	4K entries, 4-way set associative
Instruction Queue	Unified int/fp, 64 entries
Reorder Buffer	128 Entries
Execution BW	4 insts per cycle
L1 Icache/Dcache	16KB, 4-way set assoc., 64B blocks, 8 MSHRs Inst: 1 cycle latency Data: 3 cycle latency
L2 Unified Cache	1MB, 4-way set assoc. 64B block size, 10 cycle latency, 32 MSHRs
L1 to L2 cache BW	64 bytes per CPU cycle
L2 Cache to memory controller BW	4 bytes per CPU cycle
HyperTransport	8 bytes, 800 MHz
CSA Bus	8 bytes, 250 MHz
Main Memory	40 ns access latency

In order to accurately model latencies to uncacheable Ethernet device registers, we measured the latency to access the control registers on an Intel Pro/1000MT Server Adapter installed in a 2.8 GHz Pentium 4 system with an i850 chipset. We inserted a code segment into the device driver that used the Pentium 4 times-

tamp counter (using the `rdtsc` instruction) to time a load to a device register. A pair of `cpuid` instructions were used to serialize execution, guaranteeing that the load and the `rdtsc` instructions were executed in the desired order.

The ethernet card was flooded with requests using netperf to simulate a saturated link. The measurements were then made over a period that included 750,000 accesses to the ethernet card's interrupt control register. Overall, measurements revealed an average 2846 cycle latency with a standard deviation of 474 cycles. Most of the accesses were between 2500 and 3000 cycles, with none smaller than 2,500. We thus added 1000 CPU cycles to each Ethernet device register access to approximate this latency.

4. Benchmarks

For our evaluation, we focused on using netperf [11], a simple network throughput and latency microbenchmark tool developed at Hewlett-Packard. We focus on two of the many microbenchmarks that netperf includes: TCP_STREAM and TCP_RR. The TCP_STREAM benchmark is designed for bandwidth testing. The netperf client opens a connection to the netserver machine and sends data at as fast a rate as possible. The other benchmark, TCP_RR, is a request/response latency test. In this test, the client sends a MTU-sized TCP packet to the server and waits for a response. When the response comes, another message is sent immediately. The round-trip latency can be calculated as the inverse of the number of requests per second.

All netperf experiments were run initially using a functional CPU model for ten seconds of simulated real time past initialization. We generated a checkpoint 2 seconds into the test. Each experiment runs from the checkpoint for 1 billion cycles using the simple CPU model to warm up the caches, then runs for 200 million cycles with the detailed CPU model. During this last phase, statistics are gathered every 20 million cycles. This gives us a total of 10 samples. Initial experiments run from checkpoints at different locations in the program indicated that there was very little variation between regions since the benchmark is totally stable. As a result, we chose to run from only one checkpoint for each of our experiments.

There are a number of parameters that modify the behavior of these microbenchmarks. The non-default parameters that we chose to set include setting a 64kB buffer size for both sender and receiver, and setting up a 256kB socket receive buffer for both client and server. The oversized receiver was chosen due to indications that some systems may have poor socket receive buffer handling and as a result get lower than expected performance [9]. We configured the benchmark to touch the data blocks before write() or after read() to more realistically mimic cache effects.

For all of these experiments, we use the standard 1500 byte maximum transmission unit (MTU). While many networking papers vary the MTU in their experiments, 1500 bytes is the obvious first choice given its prevalence in Internet.

The inner loop in netperf is very short, basically filling up a buffer and calling the write syscall. This benchmark consequently has very little user time, and spends most of its time in the TCP/IP protocol stack of the kernel or in the idle loop waiting for DMA transactions to complete.

5. Preliminary Results

The first step of our experimentation was to use our simulator to look at the popular 1bps per 1Hz rule of thumb used in the networking community. Figure 3 and Figure 4 show this relationship for the streaming benchmark and request/response benchmark respectively. Unfortunately, limitations in the netperf streaming benchmark don't permit us to present statistics for the transmitter and as a result, we only present data for the receiver.

In both cases, we see that our experiments come close to the 1/1 ratio, though in both cases, as the clock rate is increased, the ratio drops by over half even though the bandwidths of the various system components in all configurations are adequate for moving 10Gbps over the ethernet. The inability of the 10GHz CPU to maintain the same levels as the 4GHz CPU is due to the fact that the dominant bottleneck in the benchmark is not the CPU. Though the CPU is not the major bottleneck, the direct DMA to cache configuration displays larger gains at 10GHz than at 4GHz. This is due to the fact that the cache speeds up along with the CPU, allowing the NIC to transfer data between the NIC and cache more quickly.

Figure 3 indicates some counterintuitive results First, the HTX configuration does better than the OCM configuration. One would expect to see the opposite effect since the bandwidth is higher and the latency is lower in the OCM case. We have seen in some experiments that higher bandwidths and lower latencies can cause lower performance because the lower latency to processing causes the system to process fewer packets per read system call. The increase in system calls and associated processing increases the overhead to the point where it actually reduces the system performance. Further work needs to be done here to verify if this is indeed the source of the performance loss. If this is found to be the cause, it would make sense to implement a mechanism for throttling the system call rate

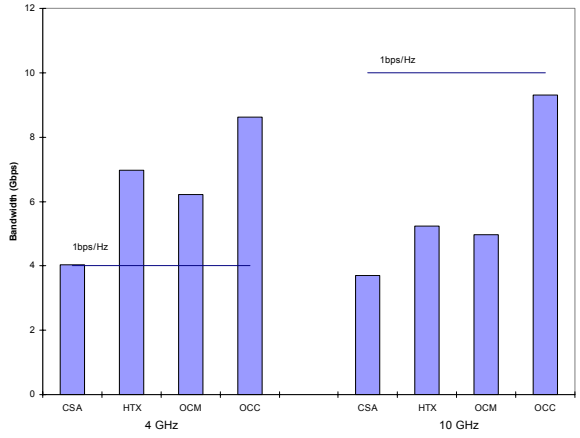

Figure 3: Streaming benchmark bandwidth

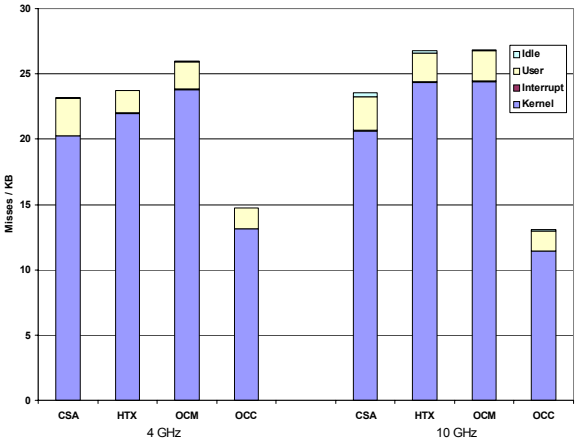

Figure 5: Streaming benchmark L2 Cache Misses/KB

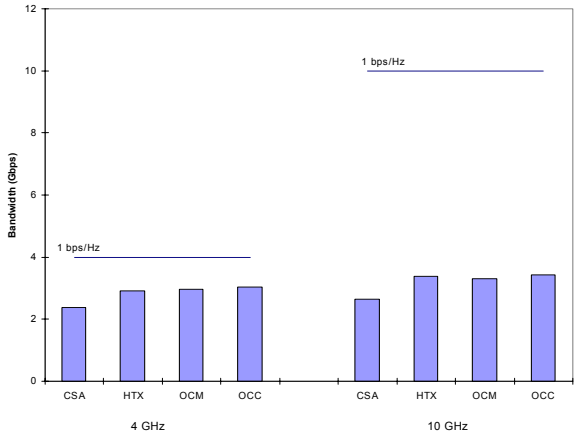

Figure 4: Request/Response benchmark bandwidth

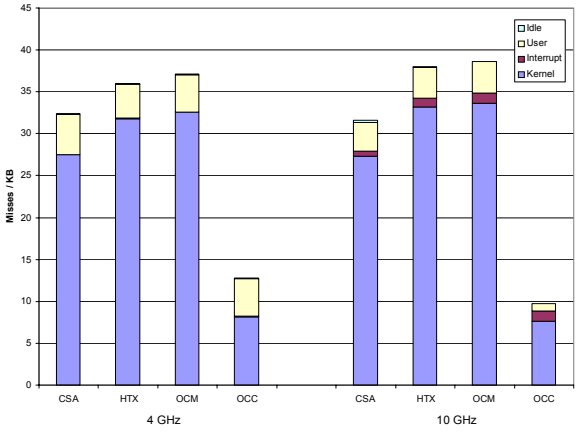

Figure 6: Request/Response benchmark L2 Cache Misses/KB

that is analogous to how interrupt coalescing limits the impact of interrupts on the system. The second counter-intuitive result can be seen in the drop in performance for CSA, HTX, and OCM when going from 4GHz to 10GHz. We haven't discovered the cause of this result and are still investigating.

The bps/Hz ratio shown in Figure 4 is for only one half of the connection. With that in mind, the ratios are similar to that of the streaming benchmark. However, in this case, the OCM configuration does better than the HTX configuration due to the fact that it is limited to only one packet in each direction per transaction. As a result, the decreased latency of the OCM case improves performance, albeit not as much as one might expect.

Figure 5 and Figure 6 show the number of cache misses involved in transmitting each KB worth of data. In the streaming case, given that we are using 64B cache lines, missing on every byte transferred would cause approximately 16 cache misses. We observe that for the benchmarks where the NIC is doing DMA into memory, these cache misses are all taken as required, though as expected, the configuration where the NIC does DMA directly to the cache, there are fewer than 16 cache misses per kB. For the streaming benchmark, the difference between the configurations that DMA to the cache and those that do not account for all 16 of the cache misses that one might expect to save. In the request/response benchmark, we see that more than 16 cache misses are saved. Presumably, some useful data was replaced by the incoming packet data or some of the packet data was replaced before it could be used.

6. Related Work

Closer integration of network interfaces with CPUs has been a prominent theme of work in the area of fine-grain massively parallel processors (MPPs). Henry and Joerg [10] investigated a range of placement options, including on- and off-chip memory-mapped NICs and a

NIC mapped into the CPU's register file. Other research machines with tight CPU/network integration include *T [19] and the J-Machine [6]. Mukherjee and Hill [18] propose several optimizations for NICs located on the memory bus that can participate in cache-coherent memory transactions. Many of their optimizations could be used with our integrated NICs as well. We see our future work, in part, as an attempt to apply some of the ideas from this custom MPP domain to the more commercially significant area of TCP/IP networking. Now that TCP/IP over off-the-shelf 10 Gb/s Ethernet can provide bandwidth and latency competitive with, and often better than, special-purpose high-speed interconnects [9], a single efficient network-oriented device with integrated 10 Gb/s Ethernet could serve as both a datacenter server part and a node in a high-performance clustered MPP.

Integrated NIC/CPU chips targeted at the embedded network appliance market are available (e.g., [3]); this work differs in its focus on integrating the NIC on a general-purpose end host, and on performance rather than cost effectiveness.

As discussed in the introduction, network interface research in the TCP/IP domain has focused on making NICs more powerful rather than bringing them closer to the CPU. To the extent that these ideas address overheads that are not directly due to the high latency or low bandwidth of communicating with the NIC—such as data copying [5, 7] and user/kernel context switching [4, 8, 23]—they could conceivably apply to our integrated NIC as well as to a peripheral device. However, most or all of these benefits can also be achieved by dedicating a host processor to protocol processing [20] rather than pushing intelligence out to an off-chip NIC, with the advantage that protocol processing will enjoy the performance increases seen by the general-purpose CPU market without effort from the NIC vendor. An integrated NIC argues even more strongly for software innovation within the scope of a homogeneous SMT/SMP general-purpose platform rather than dedicating specialized compute resources to the NIC.

Other proposed NIC features that directly address NIC communication overheads would largely be obviated by an on-chip NIC. For example, schemes that use NIC-based DRAM as an explicit payload cache [13, 24] would be unnecessary, as an integrated NIC would share the same high-bandwidth channel to main memory as the CPU. Payloads with temporal locality could even benefit from the CPU's on-chip cache hierarchy, though as discussed in Section 2, care must be taken to avoid pollution.

7. Conclusions and Future Work

We have simulated the performance impact of integrating a 10 Gbps Ethernet NIC onto the CPU die, and find that this option provides higher bandwidth and lower latency than even an aggressive future off-chip implementation. While this result may not seem surprising in retrospect, we believe the concept of CPU/NIC integration for TCP/IP processing points the way to a large number of potential optimizations that may allow future systems to cope with the demands of high-bandwidth networks. In particular, we believe this avenue of integrating simpler NICs should be considered as an alternative to the current trend of making off-chip NICs more complex.

One major opportunity for an on-chip NIC lies in closer interaction with the on-chip memory hierarchy. Our results show a dramatic reduction in the number of off-chip accesses when an on-chip NIC is allowed to DMA network data directly into an on-chip cache.

We have begun to investigate the potential for NIC-based header splitting to selectively DMA only packet headers into the on-chip cache. Clearly there is room for more intelligent policies that base network data placement on the expected latency until the data is touched by the CPU, predicted perhaps on a per-connection basis. The on-chip cache could also be modified to handle network data in a FIFO manner [25].

Another opportunity for integration lies in the interaction of packet processing and CPU scheduling. We have observed in this work the necessity for interrupt coalescing for high bandwidth streaming, and the associated penalty for coalescing in a latency-sensitive environment. An on-chip NIC, co-designed with the CPU, could possibly leverage a hardware thread scheduler to provide low-overhead notification, much like in earlier MPP machines [6, 19].

We have also demonstrated a simulation environment that combines the full-system simulation and detailed I/O and NIC modeling required to investigate these options. We have already made this environment available to other researchers on a limited basis, and plan to make a wider public release later this year.

While a general-purpose CPU is not likely to replace specialized network processors for core network functions, this trend should allow general-purpose systems to fill a wider variety of networking roles more efficiently, e.g., VPN endpoints, content-aware switches, etc. Given the very low latencies we see for integrated NICs, we also see opportunity for using this "general-purpose" part as a node in high-performance message-passing supercomputers as well, eliminating the need for specialized high-performance interconnects in that domain.

In addition to exploring the above issues, our future work includes expanding our benchmark suite to include macrobenchmarks such as SPEC WEB99, VPN, and firewall applications and comparing the performance of an integrated NIC with a TCP offload engine (TOE). To enable some of these experiments, we are in the process of shifting our simulated platform to a modern, optimized open-source OS (Linux 2.6).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCR-0105503. This work was also supported by gifts from Intel and IBM and by a Sloan Research Fellowship.

References

[1] Advanced Micro Devices, Inc. AMD eighth-generation processor architecture. White paper, Oct. 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Hamm%er_architecture_WP_2.pdf.

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

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

[4] P. Buonadonna and D. Culler. Queue-pair IP: A hybrid architecture for system area networks. In *Proc. 29th Ann. Int'l Symp. on Computer Architecture*, pages 247–256, May 2002.

[5] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications*, 39(4):68–74, Apr. 2001.

[6] W. J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G. X. Ritter, editor, *Information Processing 89*, pages 1147–1153. Elsevier North-Holland, Inc., 1989.

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

[8] P. Druschel, L. L. Peterson, and B. S. Davie. Experience with a high-speed network adaptor: A software perspective. In *Proc. SIGCOMM '94*, Aug. 1994.

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

[10] 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.

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

[12] Intel Corp. Communication Streaming Architecture - reducing the PCI network bottleneck. White paper, 2003. http://www.intel.com/design/network/papers/252451.htm.

[13] H. Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 239–250, Oct. 2002.

[14] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[15] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *Proc. 2002 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 108–116, 2002.

[16] R. Merritt. IEEE gears up for Ethernet backplane standard. *EE Times*, Feb. 2, 2004. http://www.eetimes.com/story/OEG20040202S0004.

[17] D. Minturn, G. Regnier, J. Krueger, R. Iyer, and S. Makineni. Addressing TCP/IP processing challenges using the IA and IXP processors. *Intel Technology Journal*, 7(4):39–50, Nov. 2003.

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

[19] 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.

[20] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, Feb. 2004.

[21] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, Winter 1995.

[22] L. Schaelicke and M. Parker. ML-RSIM reference manual. http://www.cse.nd.edu/ lambert/pdf/ml-rsim.pdf.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. Fifteenth ACM Symp. on Operating System Principles (SOSP)*, pages 40–53, 1995.

[24] K. Yocum and J. Chase. Payload caching: High-speed data forwarding for network intermediaries. In *Proc. 2001 USENIX Technical Conference*, pages 305–318, June 2001.

[25] L. Zhao, R. Illikkal, S. Makineni, and L. Bhuyan. TCP/IP cache characterization in commercial server workloads. In *Proc. Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2004.