Analyzing NIC Overheads in Network-Intensive Workloads

Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi,
Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt

Abstract—Keeping up with modern high-bandwidth networks is a significant challenge for system designers. A key obstacle to high network throughput is the high overhead of communication between the CPU and network interface controller (NIC), which typically resides on a standard I/O bus with high access latency. We investigate the impact of this overhead by analyzing the performance of hypothetical systems in which the NIC is more closely coupled to the CPU, including integration on the CPU die, using three network-intensive benchmarks. We find that systems with high-latency NICs spend a significant amount of time in the device driver. NIC integration can substantially reduce this overhead, providing significant throughput benefits when other CPU processing is not a bottleneck. NIC integration also enables cache placement of DMA data; this feature has tremendous benefits when payloads are touched quickly, but can harm performance in other situations due to cache pollution.

Index Terms—Hardware/software interfaces, I/O and data communications, modeling of computer architecture, simulation.

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 means that network I/O can no longer be considered an afterthought in computer system design.

Nevertheless, network interface controllers (NICs) in mainstream computers continue to be treated as generic peripheral devices connected through standardized I/O busses. While the PCI-X 2.0 and PCI Express standards are addressing this raw bandwidth mismatch of 10Gbps Ethernet (20Gbps full-duplex) and PCI-X (8.5Gbps), these 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).

One approach to addressing this bottleneck is to optimize the interface between the NIC and the CPU [3, 4, 7, 8, 15]. 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 [1]. These schemes address the I/O bus bottleneck by having the CPU interact with the NIC at a higher semantic level, which reduces the frequency of interactions. Some researchers have addressed I/O bus bandwidth by using NIC memory as a cache, eliminating bus transfers [16, 10].

This paper examines the impact of NIC access latency directly by modeling systems in which the NIC is moved progressively closer to the CPU, including integration directly on the CPU die.

- *N.L. Binkert, L.R. Hsu, A.G. Saidi, R.G. Dreslinski, and S.K. Reinhardt are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: {binkertn, hsul, saidi, rdreslin, stever}@eecs.umich.edu.*
- *A.L. Schultz is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720-1770. E-mail: alschult@eecs.berkeley.edu.*

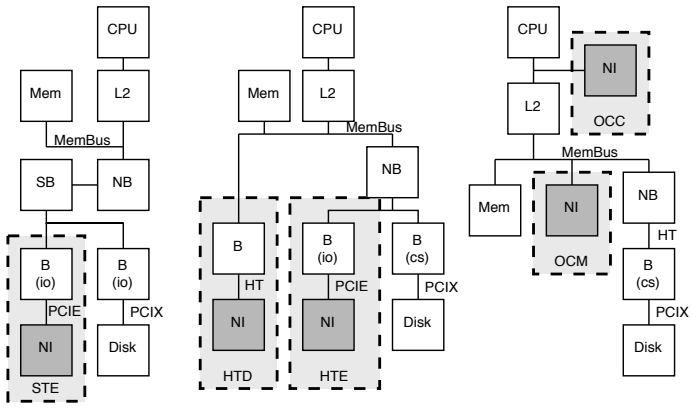

Figure 1: NIC placement options.

We model these hypothetical systems using detailed full-system simulation, and evaluate them under three network-intensive scenarios: a web server, an NFS server, and a network address translation (NAT) server. We find that systems with high-latency NICs spend a significant amount of time in the device driver. NIC integration can substantially reduce this overhead, providing significant throughput benefits when other CPU processing is not a bottleneck. NIC integration also enables cache placement of DMA data; this feature has tremendous benefits when payloads are touched quickly, but can harm performance in other situations due to cache pollution.

This paper begins with a discussion of the options for NIC placement (Section 2), followed by a description of our simulation environment in Section 3 and our benchmarks in Section 4. Section 5 presents simulation results, and Section 6 presents our conclusions and future work.

2 NIC Placement Options

To investigate the impact of changing the location of the NIC in the memory hierarchy, we chose a set of five configurations (Figure 1). The first system, standard PCI Express (STE), has an off-chip memory controller and a dedicated PCI Express x4 channel for the 10GigE NIC hanging off an I/O bridge. The HyperTransport PCI Express (HTE) configuration represents a similar system with an on-chip memory controller. In this case, the sys-

tem has one fewer chip separating the NIC from the CPU. STE and HTE are both aggressive I/O designs expected to be common in the near future. The HyperTransport direct (HTD) configuration represents a potential design for systems that implement standard I/O high speed interconnection with the NIC attaching to one of these ports. This would be analogous to attaching a NIC directly to a 6.4GB/s HyperTransport channel. The fourth and fifth configurations, on-chip memory-bus-attached (OCM) and on-chip cache-attached (OCC) represent integrating the NIC onto the CPU die. There are systems in existence today that have an integrated NIC, but we are specifically interested in the case where the NIC has direct access to the high speed memory and cache busses. The fourth design attaches the NIC directly to the memory bus to provide very high bandwidth and low latency while the fifth goes one step further and attaches the NIC to the bus between the CPU core and its last level cache (LLC). Though no detailed analysis was provided, other researchers have shown [12] that the final configuration provides the potential vast performance improvements. These improvements can be attributed to decreased latency and fewer memory bus crossings. In addition, on the receive side, direct attachment eliminates cold cache misses that are generally necessary when dealing with received data because data is provided directly to the LLC. It is important to note that there is of course a potential downside to placing data directly in the cache—pollution. We present a variation on the final two configurations, on-chip split (OCS), where the NIC is attached to both the cache bus and the memory bus. In this setup, we provide header data directly to the cache and the payload data to memory in hopes of attaining the best of both worlds.

3 Simulator Platform

Evaluation of alternative NIC architectures is usually done by emulation on a programmable NIC or by hardware prototyping. While these approaches allow modeling of different NICs in a fixed system architecture, unfortunately they do not lend themselves to modeling a range of system architectures as we have described in the previous section. We thus turn to the M5 simulator[1] for our investigation [2]. The M5 simulator is capable of booting unmodified Linux kernels; in particular, we used Linux 2.6.8.1 for this work.

M5 has a detailed CPU timing model to capture the primary timing impact of system-level interactions, including properly modeling memory barrier instructions, uncached memory accesses (e.g. for programmed I/O) and executes actual Alpha PAL code to handle interrupts and traps. To provide deterministic, repeatable simulation of network workloads, as well as accurate simulation of network protocol behavior, M5 models multiple systems and the network interconnecting them in a single process. We also use a detailed and realistic memory model capable of modeling busses of different bandwidths as well as store and forward bridges with different latencies.

The Ethernet NIC model used in M5 is based on the National Semiconductor DP83820 Gigabit Ethernet device with some minor bug-fixes and an MTU of 1500 bytes. We found we had to implement interrupt coalescing to reduce the actual number of interrupts posted so as not to overwhelm 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 (10us in our experiments).

1. http://m5.eecs.umich.edu provides more details

Table 1: Simulated System Parameters

Frequency	4 GHz, 6 GHz, 8 GHz, or 10 GHz
Fetch Bandwidth	Up to 4 instructions per cycle
Branch Predictor	Hybrid local/global (ala 21264).
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, 16MSHRs Inst: 1 cycle hit latency Data: 3 cycle hit latency
L2 Unified Cache	4MB or 16MB, 8-way set assoc. 64B block size, 25cycle latency, 40 MSHRs
L1 to L2	64 bytes per CPU cycle
L2 to Mem Ctrlr	4 bytes per CPU cycle
HyperTransport	8 bytes, 800 MHz
Main Memory	65 ns latency for off-chip controller, 50 ns on-chip

Table 2: Uncached access latencies
latency ± standard deviation (all in ns)

Device Location	Alpha DP264	Pentium III 933MHz	Pentium 4 3GHz	Simulator
Peripheral	788 ± 40	835 ± 54	803 ± 24	773 ± 8.6
Off NB	—	423 ± 49	392 ± 21	381 ± 7.2
On NB	475 ± 26	413 ± 61	216 ± 16	190 ± 2.0
On Die	—	132 ± 52	82 ± 3	30 ± 2.9

The parameters we used in modeling the configurations of Section 2 are listed in Table 1. We are modeling the approximate characteristics of these systems, rather than absolute performance, so some of these parameters are approximations. In addition to the parameters shown, it is worth noting that we also add a bridge-dependent latency penalty for each bus bridge that connects devices on separate chips. These bridge latencies were tuned based on measurements taken from real hardware using a custom written Linux kernel module. Table 2 presents timings for devices on three real machines in four different locations corresponding to NIC locations that we studied.

The peripheral device, similar to the PCIE configuration, represents a modern device hanging off a commodity I/O bus, in which multiple bridges need to be crossed to access the device. The off north bridge (NB) location is similar to our HTE configuration, where a standard I/O bus is connected to the NB directly. Similarly the HTD configuration could be realized by physically integrating the NIC into the NB. Thus we used the access latency comparable to that of an integrated NB device.

OCM is represented by measurements for a device integrated on the CPU die. The Alpha EV6 has no such devices, but both Pentium chips have an integrated local I/O APIC. These devices have an access time of 3x-4x more than the on-chip access time we modeled. However, we believe these devices were not designed to minimize latency, and that an integrated NIC could be designed with an access time closer to that of a cache.

We measured the memory access latencies for our configurations with an on-chip and off-chip memory controller. With our memory configurations an on-chip memory controller has as

access latency of 50ns and an off-chip memory controller has a latency of 65ns. In both cases our results are similar to the published [11] numbers of 50.9ns and 72.6ns respectively.

4 Benchmarks

For our evaluation, we used several standard benchmarks: netperf [9], a modified SPECweb99 [14], and NFS with a Bonnie++ [5] stressor. All benchmarks were simulated in a client-server configuration. The system under test was modeled in detail, with detailed CPU, timing, and memory modeling. The other system was run artificially quickly (simple in-order functionally modeled CPU with perfect memory system) to keep it from being the bottleneck in the simulation. In addition to the simple client-server setup, the netperf and SPECweb99 benchmarks were simulated in a network address translation (NAT) configuration with the client system accessing the server through a NAT box.

Netperf is a simple network throughput and latency microbenchmark tool developed at Hewlett-Packard. We focus on two of the many microbenchmarks that are included in netperf: stream, a transmit benchmark; and maerts, a receive benchmark. In both of these, the netperf client opens a connection to the machine running the netserver and sends data at the highest rate possible.

SPECweb99 (Specweb) is a well-known webserver benchmark that stresses the performance of a server machine. Simulated clients generate web requests to form a realistic workload for the server, consisting of static and dynamic GET and POST operations over multiple HTTP 1.1 connections. Specweb also involves dynamic ad rotation using cookies and table lookups. The original benchmark is intended to be tuned to determine the maximum number of simultaneous connections a server is able to handle. However, iterative tuning is impractical for our purposes due to the relative slowdown of our simulated environment. We created our own benchmark client that generates Specweb client requests using the same statistical distribution as the original clients, but without the throttling. Our version thus continuously generates packets until the link bandwidth is saturated. We use the Apache http server, version 2.0.52, with a maximum of 100,000 clients and 50 threads per child.

NFS is a network file system from Sun. We ran Bonnie++, a simple benchmark for testing hard drive and file system performance, on our client to exercise a remote NFS server. The Bonnie++ tests we utilized consist of a series of block writes across the network.

For our NAT configurations, we simply placed another machine between the server and client to act as a NAT machine. The client's requests are translated by the NAT machine from a private network address to the address of the NAT machine, so that the client's true IP address is hidden from the server by the NAT box. When conducting these experiments, the system under test was the NAT machine.

The results we present in the next section were composed from a single sample of 200 million cycles from the above mentioned benchmarks. However, we did run many of the configurations for an extended period of time taking 50 or more samples of 200 million cycles each. These results show a coefficient of variation of less than 10% for all metrics we discuss in this paper. Furthermore if we increase the cycle time to 400 million cycles the coefficient of variation decreases to around 5%. We consider these results stable and intend to re-run all our numbers using a 400 million cycle sample time in the near future. An exception to this is our NFS benchmark which does not run long enough to take as many samples as we would like.

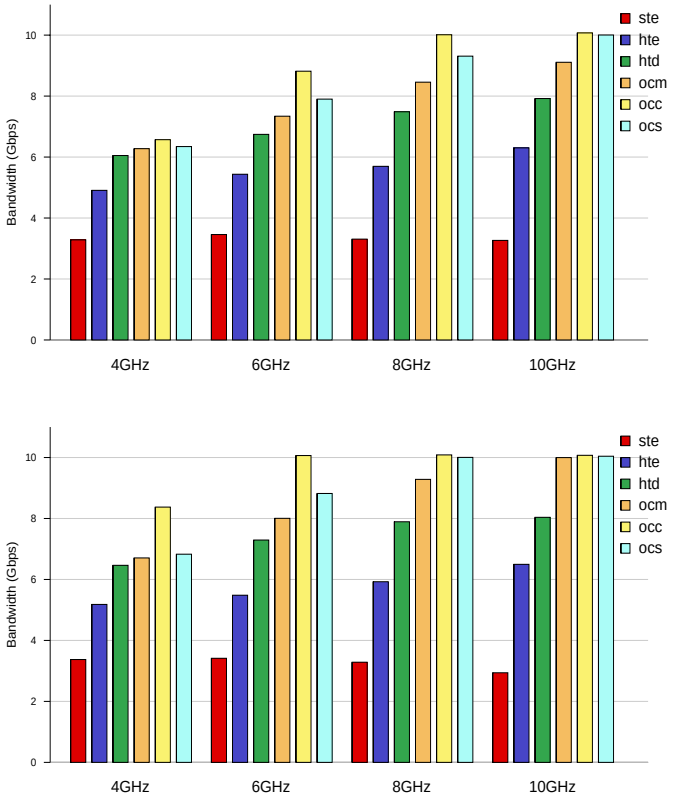

Figure 2: TCP receive microbenchmark performance. (Top: 4MB, Bottom: 16MB)

5 Results

We first examined the behavior of our simulated configurations using the netperf microbenchmark, varying CPU speed and last-level cache (LLC) size. We then use our application benchmarks (web, NFS, and NAT) to explore the performance impact of our system configurations under more realistic workloads.

5.1 Microbenchmark Results

Figure 2 plots the achieved bandwidth of our TCP receive microbenchmark across our six system configurations, four CPU frequencies (4, 6, 8, and 10 GHz) and two cache sizes. Although the higher CPU frequencies are not practically achievable, they show the impact of reducing the CPU bottleneck, which will likely be achievable through multiprocessing (at least for the macrobenchmarks below which exhibit easily exploited connection-level parallelism). The integrated NICs universally provide higher performance, though their advantage over the direct HyperTransport interface is slight at lower frequencies when the benchmark is primarily CPU-bound. The more tightly coupled interfaces also get more of a boost from larger LLC sizes. In some situations, the in-cache DMA configuration (OCC) provides

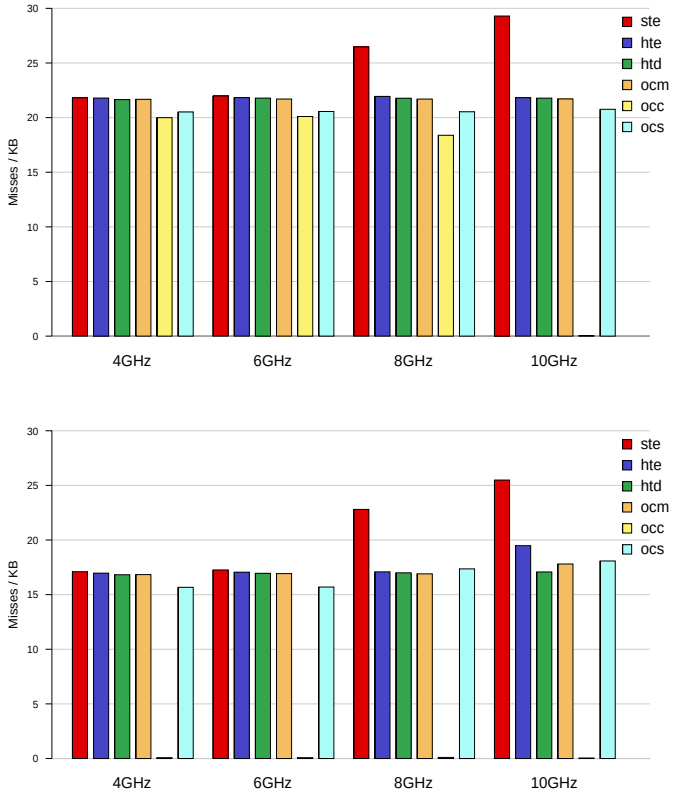

Figure 3: TCP receive microbenchmark misses per kilobyte. (Top: 4MB, Bottom: 16MB)

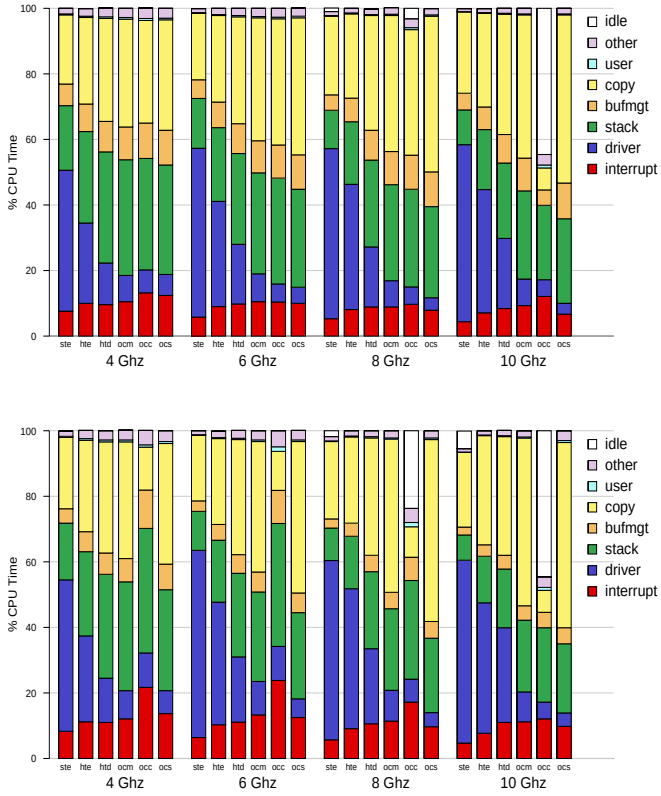

Figure 4: TCP receive microbenchmark CPU utilization. (Top: 4MB, Bottom: 16MB)

higher performance than OCM and OCS. The explanation for this difference can be seen in Figure 3, which shows the number of last-level cache misses per kilobyte of network bandwidth for these configurations. When the cache is large enough to hold all of the network receive buffers, OCC dramatically reduces the number of cache misses incurred. Interestingly, this condition is a function of both the cache size and the CPU speed: a faster CPU is better able to keep up with the network and thus its buffers fit in a smaller cache. Because our microbenchmark does not perform any application-level processing, the cache pollution induced by OCC when the cache is too small does not negatively impact performance. We will see a counterexample when we look at macrobenchmarks below.

Figure 4 breaks down CPU utilization for the same configurations just described. Clearly, moving the NIC closer to the CPU drastically reduces the amount of time spent in the driver, as it reduces the latency of accessing device registers. This translates directly to the higher bandwidths of Figure 2. However, most cases are still CPU-bound, as can be seen by the idle times shown. Only with the OCC configuration are we able to make even the 10GHz CPU powerful enough to cease being the bottleneck, as can be seen by its much increased idle time percentage. OCC reduces the time spent in copy, since the source locations of data from the network are in the cache rather than in memory. Though this reduction occurs in all configurations, it is not uniform across all CPU speeds. This is because copy time goes hand in hand with misses/kB -- when there is a miss in the cache, you must copy the data in from memory. Looking at all three graphs together tells you that OCC at 10GHz saturates the network at 10Gbps, as some other configurations also manage to do, but only OCC at 10GHz has idle capacity in the CPU to perform other tasks.

Figure 4 also illustrates a potential pitfall of integration: overresponsiveness to interrupts. Because the CPU processes interrupts much more quickly with the on-chip NIC, it processes fewer packets per interrupt, resulting in more interrupts and higher interrupt overhead.

Figure 5 presents the performance, cache, and cpu utilization results for the TCP transmit microbenchmark at a 4MB last-level cache size. For this microbenchmark, we configured the sender to not touch the payload data before it is sent. The result here is similar to what one might see in a web-server where the content is static. Since the payload data is not touched, a larger cache size does not change the results as compared to a 4MB LLC, making presentation of these results unnecessary.

At low frequencies, on-chip NICs exhibit a noticeable performance improvement over direct HyperTransport; because transmit is interrupt intensive, low-latency access to the NIC control registers speeds processing. Again, we see that faster processors increase the utility of in-cache DMA, as they have fewer outstanding buffers and are thus more likely to fit them all in the cache. Although all of the configurations have some idle time, with the faster CPUs the on-chip NICs have a distinct advantage over HTD. When looking at the cache performance results, the bulk data transfer is done by DMA reads rather than writes, DMA data placement affects only headers and acknowledgment pack-

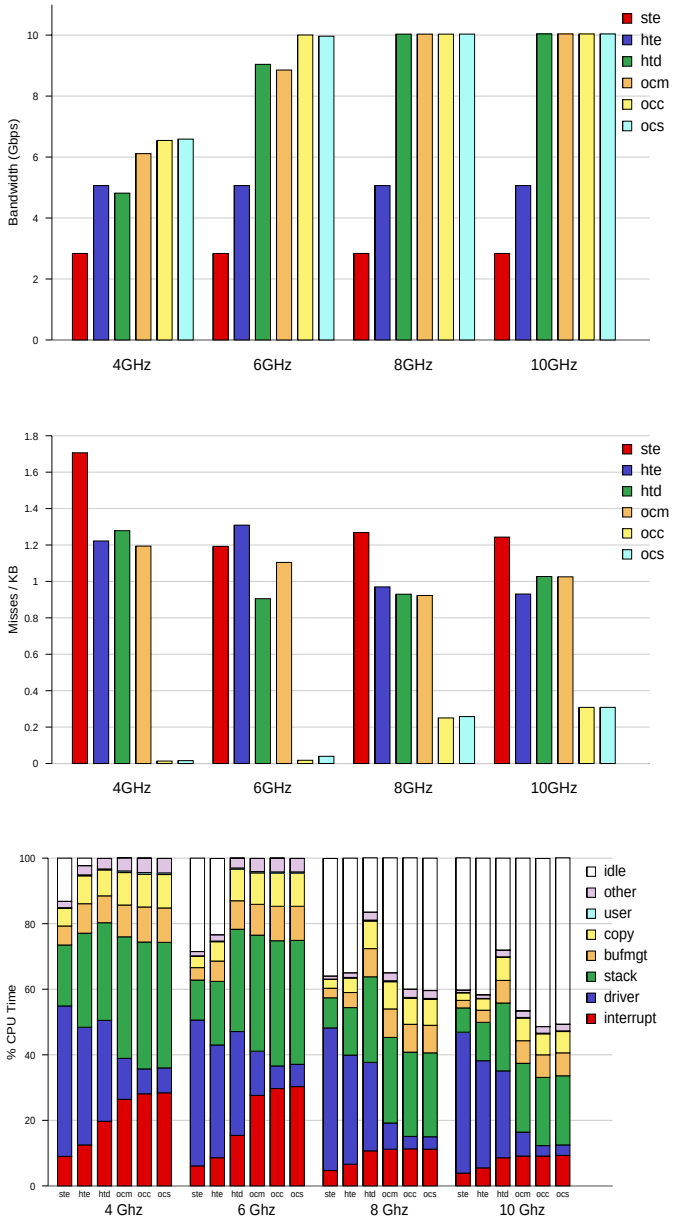

Figure 5: TCP transmit microbenchmark results.

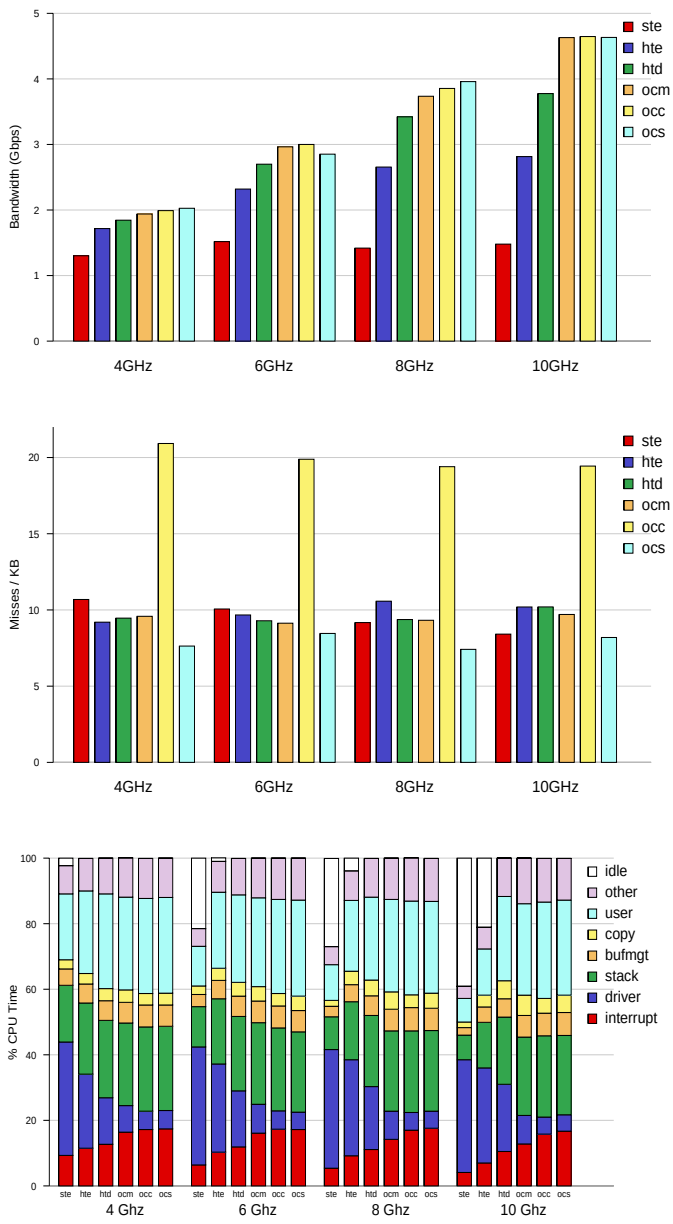

Figure 6: Web server benchmark.

ets, giving OCC and OCS similar performance, but only slightly ahead of OCM.

The high idle time in STE and HTE is due to poor overload behavior; note that the link bandwidth is only a fraction of what HTD and the on-chip interfaces achieve. We are investigating whether this behavior is due to our device model, the NS83820 driver, or is inherent in Linux 2.6.

5.2 Macrobenchmark Results

While the microbenchmark results provide valuable insight into the fundamental behavior of our configurations, they do not directly indicate how these configurations will impact real-world performance. To explore this issue, we ran the three application-level benchmarks described in Section 4: the Apache web server, an NFS server, and a NAT gateway. Although we ran with both 4 MB and 16 MB caches, we present only the 4 MB results here. For each benchmark, we show network throughput, L2 cache misses per kilobyte of network data transmitted, and a breakdown of CPU time.

The web server results are shown in Figure 6. In this test, we can see that the 4 GHz runs are CPU limited and only very minor performance improvements are realized by tighter integration of the NIC. On the other hand, the 10 GHz runs are network bound and achieve marked improvement in bandwidth when the NIC is tightly integrated. While a 10 GHz CPU may never be realized, a webserver benchmark is highly parallel, and this single-cpu 10 GHz system performance could readily be achieved by a chip multi-processor system. Another thing that stands out in these graphs is that OCC has the opposite effect of misses/kB with the webserver benchmark as compared to the microbenchmarks. This

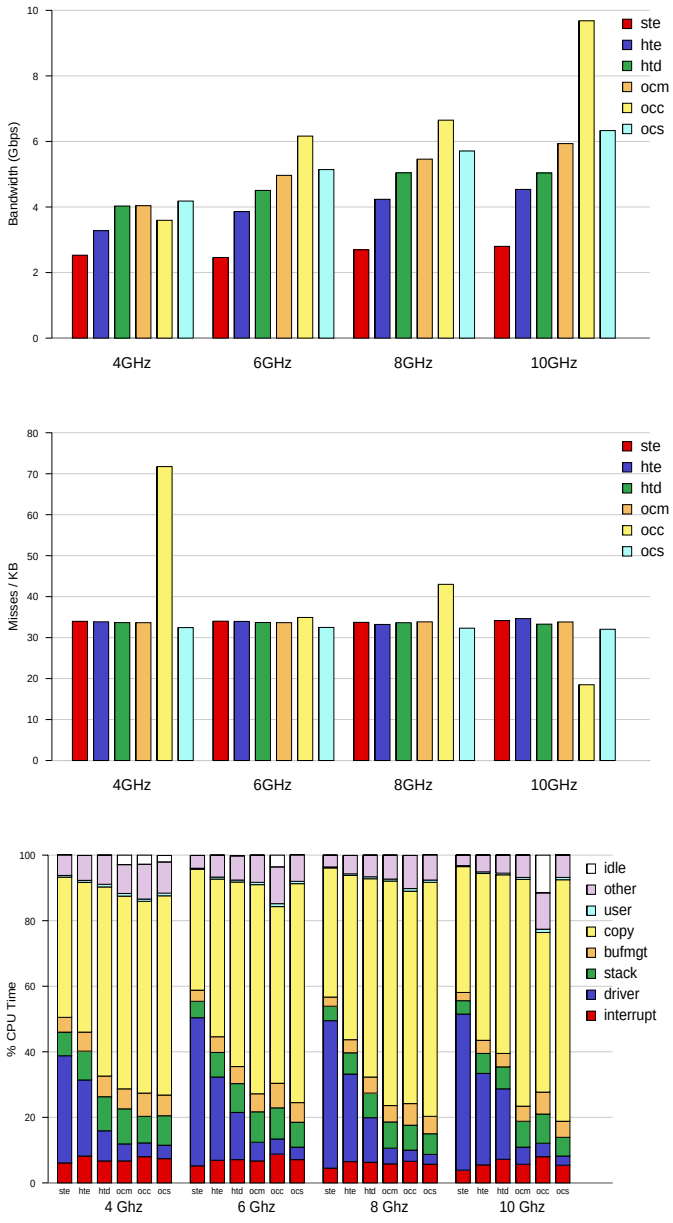

Figure 7: NFS server benchmark

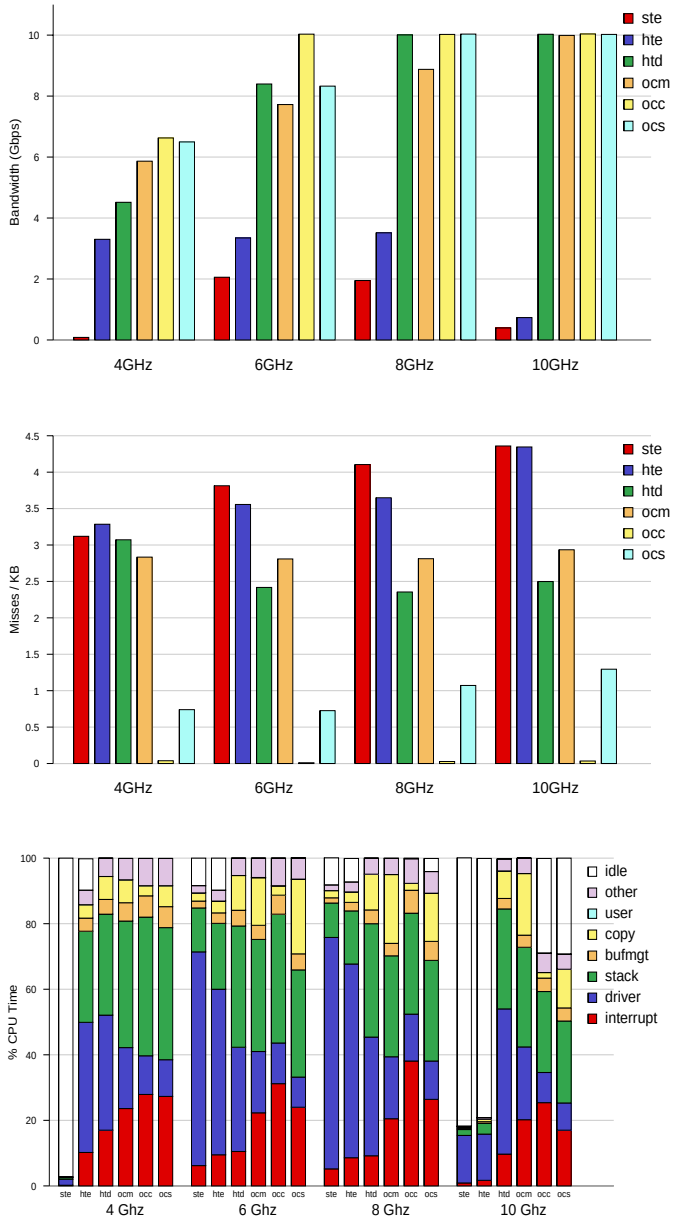

Figure 8: NAT gateway benchmark:

is unsurprising since the working set of this application is actually non-trivial, unlike the microbenchmarks. Thus, the OCC configuration actually pollutes the cache and reduces cache performance. One thing that is important to note is that in this case, the misses occur to userland data not related to the networking. This is evidenced by the fact that the fraction of time spent copying does not increase even though misses per kilobytes increases. Because of the pollution, it is clear that OCS, which is the hybrid OCC/OCM configuration, achieves the lowest cache miss rate out of all configurations.

Figure 7 shows the NFS server for the various configurations. Again, the 4 GHz runs are largely CPU bound and do not exhibit significant performance improvement with the on-chip NICs. Here, the interplay between network buffer sizing and CPU speeds is clearly illustrated. When looking at the bandwidth and misses/kB graphs, OCC clearly just pollutes the cache. However, at 10GHz, despite having the same cache size, OCC is a boon to performance. This has to do with the rate at which the CPUs can pick up packets for processing. Since the 10GHz machine is so fast, the buffering it requires is less, thus the appearance of working set sizes changing with CPU speed. As with the microbenchmark, moving the NIC closer to the CPU drastically reduces the amount of time spent in the driver since it reduces the latency of accessing device registers. In addition, the time spent copying is similarly proportional to the misses. In nearly all cases, these effects result in improved bandwidth due to the loosening of the CPU bottleneck.

Figure 8 shows the NAT gateway performance. In this case, we are running the TCP receive microbenchmark between two hosts on either side of the gateway. The poor performance in the low

performance configurations is due to poor behavior under overload conditions. The insertion of a NAT machine that is modeled in detail between a fast server and client is potentially a major cause of network congestion in our simulation. The configurations with unexpectedly significant idle time in their NAT runs do have packets dropped at the NAT box, and we believe the poor performances are due to the TCP stack attempting congestion control at the endpoints. But our runs do not run long enough to achieve steady state to confirm this, and we are still looking into other possibilities. However, this hypothesis makes sense since the faster CPUs, coupled with the low-latency NIC placements do not have trouble, since the increased speeds allow the NAT machine to not drop packets.

The misses/kB graph shows that the OCC configuration eliminates all misses, while the OCS eliminates misses on only header data, as we had hoped for. Also note that for this configuration, OCC and OCS are able to saturate the network link while having CPU to spare.

Overall, tighter integration is clearly a performance win in all cases, but the best performer varies depending on the application. DMAing directly to the cache can yields huge performance differences in some circumstances while hurting performance in others. The header splitting configuration is a reasonable first step in attempting to mitigate the problem of cache pollution while achieving some of the benefit, but more can be done.

6 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. 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 [17].

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. Along with this benefit comes an 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, 13].

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 in the near future.

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 integrated NICs can achieve, 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 additional macrobenchmarks. We currently have a VPN application and an iSCSI-based storage workload under development. A comparison of the performance of an integrated NIC with a TCP offload engine (TOE) is highly desirable, but a TOE model and the associated driver and kernel modifications would be very complex to implement.

Acknowledgement

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

References

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

[2] 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*, February 2003.

[3] Philip Buonadonna and David 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.

[4] Jeffery S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications*, 39(4):68–74, April 2001.

[5] Russell Coker. http://www.coker.com.au/bonnie++/.

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

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

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

[10] Hyong-youb Kim, Vijay S. Pai, and Scott 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, October 2002.

[11] Xbit Laboritories. http://www.xbitlabs.com/articles/cpu/display/lga775_19.html.

[12] Dave Minturn, Greg Regnier, Jon Krueger, Ravishankar Iyer, and Srihari Makineni. Addressing TCP/IP processing challenges using the IA and IXP processors. *Intel Technology Journal*, 7(4):39–50, November 2003.

[13] 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.

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

[15] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner 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.

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

[17] Li Zhao, Ramesh Illikkal, Srihari Makineni, and Laxmi Bhuyan. TCP/IP cache characterization in commercial server workloads. In *Proc. Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2004.