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

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

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

Abstract

Modern high-bandwidth networks place a significant strain on host I/O subsystems. However, despite the practical ubiquity of TCP/IP over Ethernet for high-speed networking, the vast majority of end-host networking research continues in the current paradigm of the network interface as a generic peripheral device. As a result, proposed optimizations focus on purely software changes, or on moving some of the computation from the primary CPU to the off-chip network interface controller (NIC). We look at an alternative approach: leave the kernel TCP/IP stack unchanged, but eliminate bottlenecks by closer attachment of the NIC to the CPU and memory system.

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.

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 Mbps to 10 Gbps, 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, network interface controllers (NICs) in mainstream computers continue be treated as generic peripheral devices connected through standardized I/O buses. Because I/O stan-

dards evolve relatively slowly, their performance lags behind that of both CPUs and networks. For example, it is currently possible to purchase a full-duplex 10 Gbps Ethernet card capable of 20 Gbps total bidirectional network throughput. But the theoretical peak throughput is limited to 8.5 Gbps by the 133 MHz 64-bit PCI-X bus used by this card. 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. While the PCI-X 2.0 and PCI Express standards are addressing this raw bandwidth mismatch, 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 3.4).

The current industry approach to addressing this bottleneck is to optimize the interface between the NIC and the CPU [4, 5, 7, 8, 22]. 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, which reduces the frequency of interactions. More recently, some researchers have addressed I/O bus bandwidth directly, eliminating bus transfers by using NIC memory as a cache [23, 12].

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. This requires hardware companies to 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 until just a few years ago. 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 traditional NICs: closer (i.e. on-die) coupling of a conventional NIC with the CPU. Unlike adding intelligence to the NIC, this approach does not require significant software changes, making it possible to re-use existing device drivers with at most minor modifications. Although an integrated NIC provides less flexibility than an add-in card, the dominance of Ethernet, a packet based network, makes the choice of link-layer protocol a non-issue. Low power applications could integrate all of the logic onto a single chip, whereas server level single-chip multiprocessors could sport a higher bandwidth interface connecting to off chip switching as a product differentiator. High-end multi-chip systems could share integrated NICs easily across their inter-chip interconnect, similar to the manner in which AMD Opteron processors utilize their HyperTransport channels to share DRAM and I/O channels [1].

In this paper, we compare six simulated basic system configurations connected to a 10 Gbps Ethernet link. Two configurations model conventional systems with PCI Express-like I/O channels, while four explore alternative NIC locations: one directly attached to a HyperTransport-like interconnect, and three directly integrated on the CPU die. On-die NIC configurations not only benefit from lower latency communication with the CPU, but have direct access to the on-chip memory hierarchy. We explore three options for placement of incoming NIC DMA data in the

memory hierarchy: writing all data to off-chip DRAM, writing all data to the on-chip L2 cache, and a “header splitting” configuration that writes only the initial portion of each packet to the on-chip cache and sends the remainder to off-chip DRAM.

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—e.g. processors, memory hierarchy, network adapters—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, 22, 12]. Unfortunately, this approach is not feasible for our proposal because we are not modifying the NIC but rather integrating it onto the processor die itself. We have developed a simulation environment specifically targeted towards 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-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: TCP send and receive microbenchmarks, a web server, an NFS server, and a network address translation (NAT) gateway. We find that:

- There is a clear benefit to an on-die NIC in terms of bandwidth and CPU utilization, particularly on the transmit side.
- Placing NIC data in the on-chip memory hierarchy can have a significant positive impact on performance, but only for certain workloads and certain combinations of CPU speed and cache size. In the case of the web server, in-cache data placement significantly increases the cache miss rate due to pollution, but this only has a slight impact on performance. Because the NAT gateway both transmits and receives data intensively, its performance sees the greatest benefit from the combination of the on-chip NIC and in-cache data placement.
- In general, larger caches and/or faster CPUs increase the likelihood that the processor will touch network data before it is kicked out of the cache. In contrast, smaller caches and slower CPUs are less likely to do so, and thus suffer from pollution effects.
- Using a “header splitting” technique (placing headers in the cache and packet payloads directly in DRAM) reduces pollution problems, and often provides benefits by eliminating misses in the kernel stack code, but falls well short of the performance of placing full payloads in the cache when the latter technique provides a benefit.

As a result, we conclude that more intelligent policies for managing network data placement are required to get both the full benefit of payload cache placement when possible and avoid its pollution effects in other situations.

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

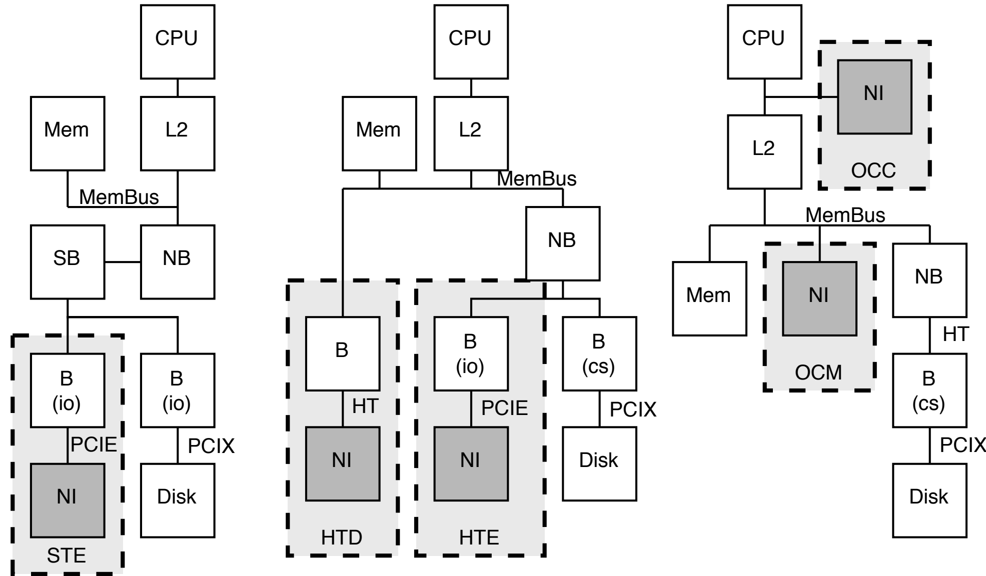

Figure 1: NIC placement options.

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 system 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 remaining configurations have the NIC integrated onto the CPU die, as we propose. We believe the area and complexity required for this integration are modest. The only required on-chip components are those which enqueue and dequeue packet data for into and out of the network FIFOs. It is not necessary to integrate of the physical layer signalling logic, or even the bulk of the FIFO buffer space. If pins are scarce, network data could be multiplexed over an I/O channel (e.g., HyperTransport). This differs from the HyperTransport-attached NIC in that CPU/NIC interactions are on-chip, and only latency-tolerant transfers between on-chip and off-chip network FIFOs go across the HyperTransport link.

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. We look at two specific on-chip configurations, on-chip memory-bus-attached (OCM) and on-chip cache-attached (OCC). OCM attaches the NIC directly to the memory bus to provide very high bandwidth and low latency while the OCC 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 [16] 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

As discussed in the introduction, 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 simulation for our investigation.

Network-oriented system architecture research requires full-system simulation, capable of running OS and application code. Additionally, it must have 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 platform 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. This includes driver code to communicate with the NIC, the network stack, and the copy code. Most network intensive applications spend the majority of their time in this kernel code rather than user-level application code; thus conventional architectural simulators, which execute only user-level application code with functional emulation of kernel interactions, do 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. Adding this functionality would not have been straightforward. To address this, we decided to extend M5, our existing application-only architectural simulator, to meet our full-system needs. M5, which has roots in the SimpleScalar [5] simulator but has since become fully unrecognizable from its ancestor, executes the Alpha ISA. Adding full-system capabilities included implementing true virtual/physical address translation, processor-specific and platform-specific control registers (enabling the model to execute Alpha PAL code), and all other Alpha Tsunami chipset and peripheral devices. The final result retained the name M5; details regarding its capabilities can be found here [2].

We used SimOS/Alpha [21] as a reference platform for development, which required us to functionally emulate the Alpha 21164 processor. 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. It also supports all user-visible 21264-specific Alpha ISA extensions. SimOS/Alpha is able to boot the DEC Tru64 Unix kernel, but we wanted to boot Linux to better facilitate kernel modifications as necessary for our research. Unfortunately, Linux does not support the SimOS TurboLaser platform, which compelled us to switch to the Tsunami chipset

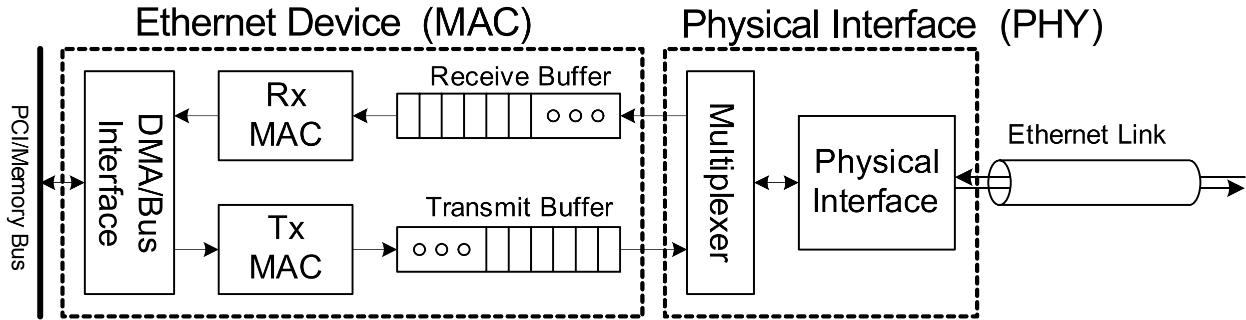

Figure 2: Network Interface Controller (NIC) device model.

instead. Our simulator is now capable of booting unmodified Linux, including kernels in the 2.4.x series and the 2.6.x series. We used Linux 2.6.8.1 to generate the results for this paper.

In addition to these modifications, we 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. Finally, 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 key determinants of networking performance. We use a handful of simple component models to construct system models representing those of Section 2.

To emulate all of the interconnects in the system, we use a single bus model of configurable width and clock speed. 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. The 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 io in Figure 1) and for the memory controller. In our model the memory controller simply bridges between two busses (for example, the front-side bus and the controller/DRAM bus). The DRAM timing is modeled in the DRAM object itself.

3.3 Ethernet Device Model

Our Ethernet NIC model, shown in Figure 2, is modeled after the National Semiconductor DP83820 Gigabit Ethernet device. Unlike a real DP83820 we allow its TX DMA engine to write to arbitrarily aligned blocks. This common feature in most gigabit Ethernet controllers allows us

to align the majority of the packet. 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.

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. Finally, the MAC implements interrupt coalescing to reduce the interrupt rate. 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. We use a fixed-delay scheme to reduce the actual number of interrupts posted, in which the device uses a timer to defer delivering a packet interrupt for a specified time (in most of our experiments, we use 10 us). 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.

The physical interface model is a functional component that moves data from the transmit buffer to the link or passes data from the link to the receive buffer. Since there is no buffering in the physical interface and it represents negligible delay, its timing impact is not modeled.

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 modeling a simple wire, only one packet is allowed to be transmitted in each direction at any given time. 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.

3.4 Memory Latencies

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. Our module used the CPU cycle counting instructions in conjunction with serialization instructions (x86) or false dependencies (Alpha) to insure that the cycle counter is not read until the access has completed.

Table 2 presents timings for devices on three real machines in six different configurations, reflecting the different NIC locations that we studied. Below is a brief description of the location of each device.

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.

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 and 16 MB, 8-way set assoc. 64B block size, 25 cycle 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

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 [15] numbers of 50.9ns and 72.6ns respectively.

4. Benchmarks

4.1 Workloads

For this evaluation, we used several standard benchmarks: netperf [13], a modified SPEC WEB99 [21], and NFS with a Bonnie++ [7] stressor. All benchmarks were simulated in a client-server configuration. The system under test was modeled with detailed CPU timing, memory timing, and peripheral device interaction. The other system(s) stressing the system under test were operated artificially quickly (functionally modeled with a perfect memory system) so as not to become the bottleneck. In addition to the simple client-server setup, the netperf and SPEC

WEB99 benchmarks were simulated in a network address translation (NAT) configuration with the client system accessing the server through a NAT gateway.

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 netperf server and sends or recieves data at the highest rate possible. The benchmark itself has a very short inner loop, 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.

SPECWEB99 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 contains 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 ubiquitous network file system from Sun that enables users to see a distributed file system network locally on their individual machines. Thus, a simple copy of one file to another location incurs network traffic. To remove the the disk subsystem from this benchmark, the system under test(server) ran with a RAM disk that it exported over NFS and the client ran Bonnie++, a simple benchmark for testing hard drive and file system performance. The Bonnie++ tests we utilized consist of a series of large block writes across the network.

For Netperf over NAT, we simply placed another machine between the server and client to act as a NAT machine. We ran both stream and maerts in this configuration. The client's requests are translated by the NAT machine from a private network addresses to the address of the NAT machine so it can be routed on the public internet. When conducting these experiments, the system under test was the NAT machine.

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

4.2 Methodology

The time required for full-system cycle-level simulation is immense and, in general, is orders of magnitude slower than a real system's performance. Thus running any benchmark to completion is impractical. To address this we used a combination of functional fast-forwarding, warm-up and sampling to obtain the results herein.

The code used to generate the executables we run as benchmarks has been modified to insert special instructions that do not exist in the Alpha ISA. These opcodes, which we call m5ops, allow a normal program to pass information to the simulator through registers to trigger when, for example, a checkpoint should be created, or statistics should be reset. To get the most interesting

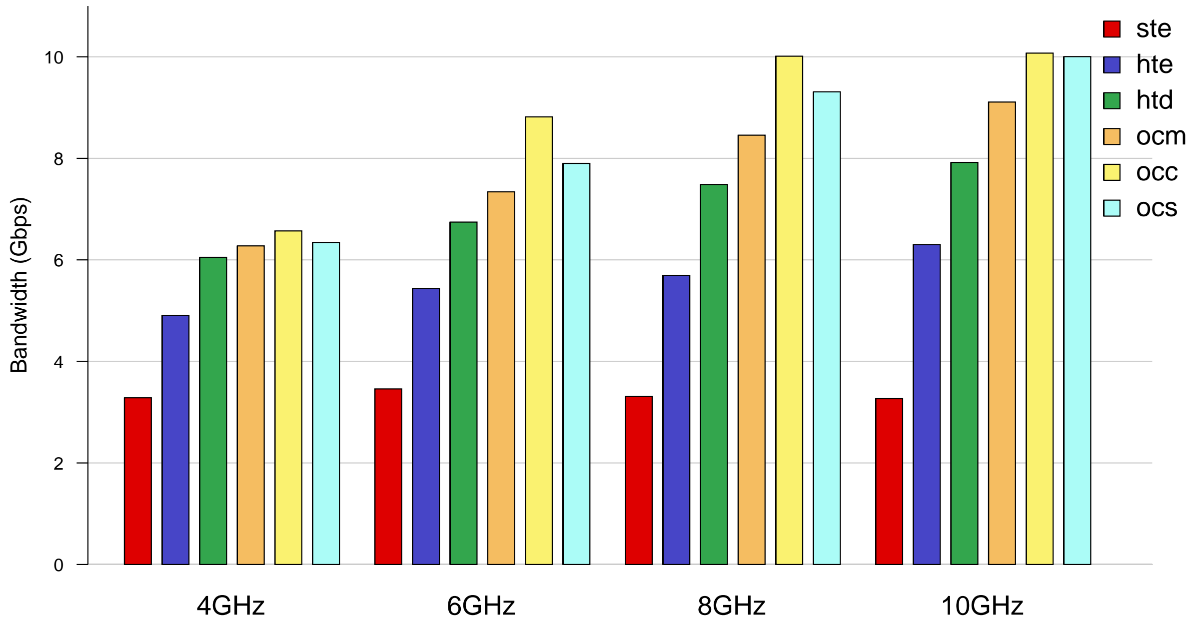

Figure 3: Achieved bandwidth for the receive microbenchmark. (4MB L2 Cache)

portion of each benchmark we have modified our binaries to create checkpoints when the workload begins its steady state processing.

From a checkpoint we warm-up the state in the caches and TLB for 200 million cycles using a simple CPU model. After the warm-up is complete we switch to a detailed cycle-level CPU model and run for another 200 million cycles.

The results presented 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 as many as 50 samples of 200 million cycles each. These runs show a coefficient of variation of 5% ± 2.5% for the macrobenchmarks and under 1% for the microbenchmarks. Furthermore if we increase the sampling time to 400 million cycles, the coefficient of variation decreases to around 5% ± 1% for the macrobenchmarks and under 0.5% for the microbenchmarks. We consider these results stable and intend to re-run all our numbers using a 400 million cycle sample in the near future. An exception to this is our NFS benchmark which does not have a single phase that runs long enough to take as many samples as we would like.

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 3 and Figure 4 plot 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

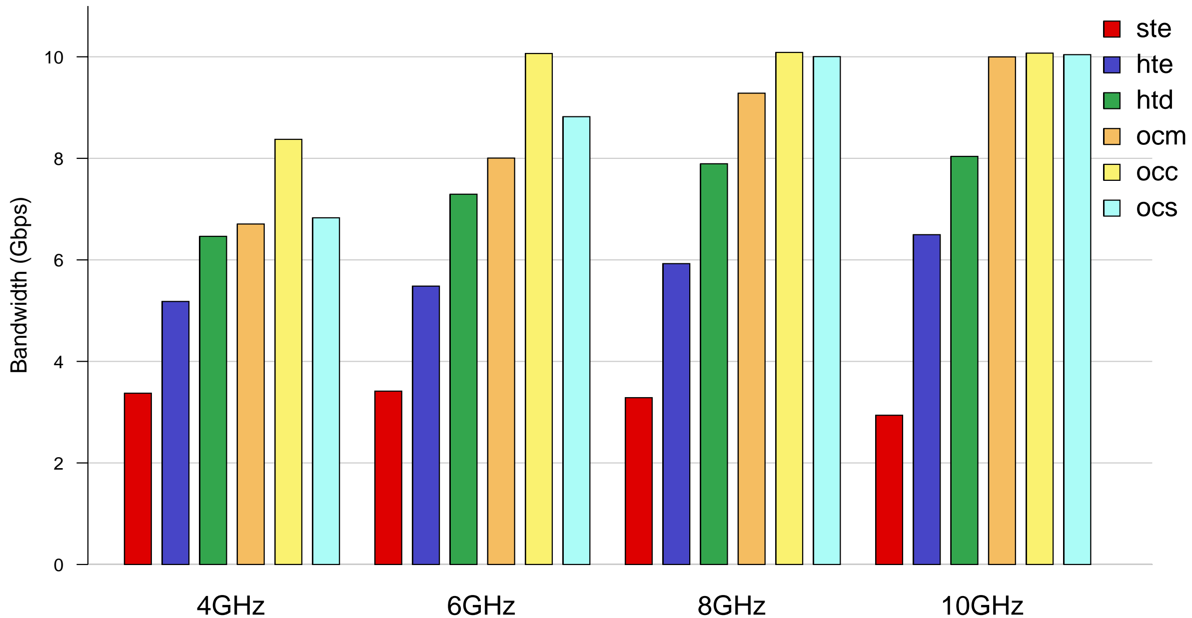

Figure 4: Achieved bandwidth for the receive microbenchmark. (16MB L2 Cache)

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 higher performance than OCM and OCS. The explanation for this difference can be seen in Figure 5 and Figure 6, 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 7 and Figure 8 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 3 and Figure 4. 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 shows that OCC at 10GHz saturates the network at 10Gbps, as some other config-

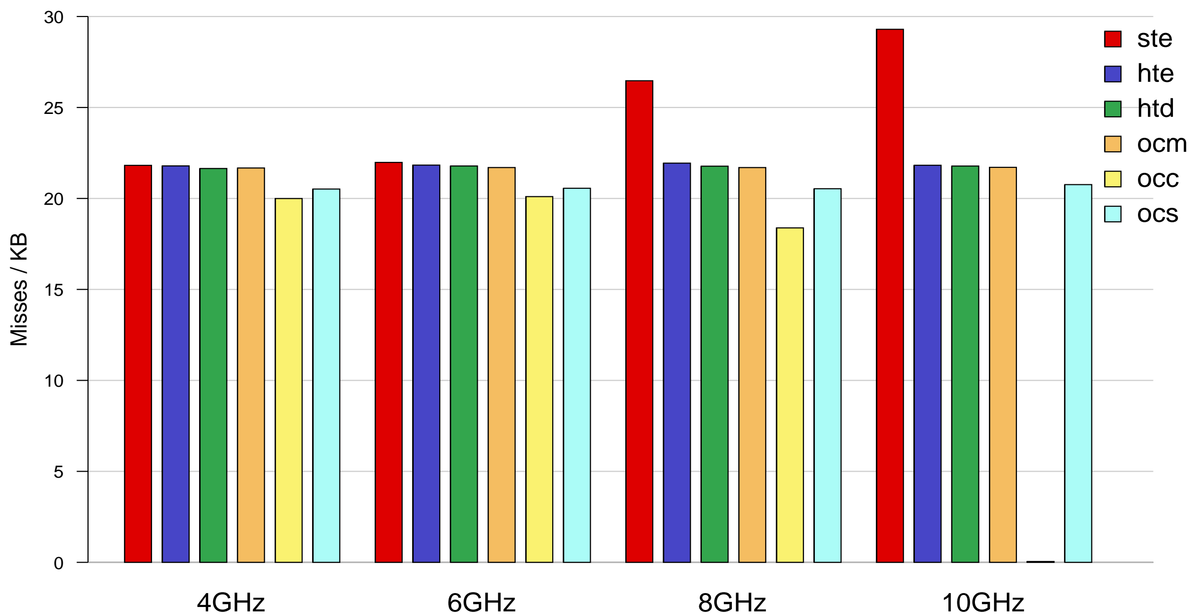

Figure 5: Misses per kilobyte for the receive microbenchmark. (4MB L2 Cache)

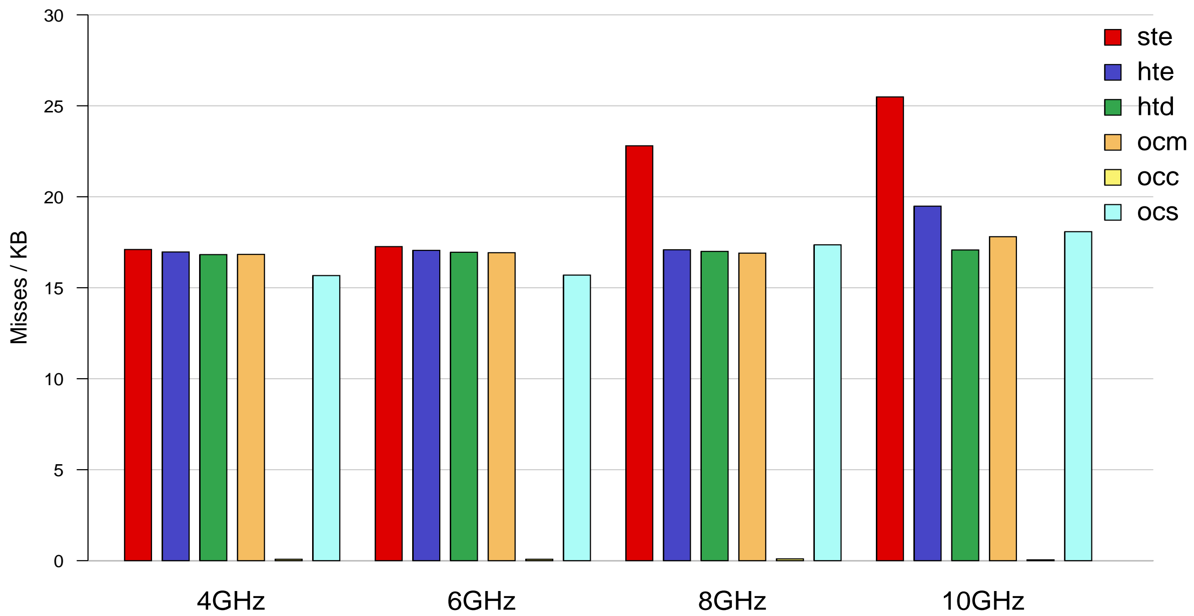

Figure 6: Misses per kilobyte for the receive microbenchmark. (16MB L2 Cache)

urations also manage to do, but only OCC at 10GHz has idle capacity in the CPU to perform other tasks.

Figure 7 and Figure 8 also illustrate a potential pitfall of integration: over-responsiveness 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.

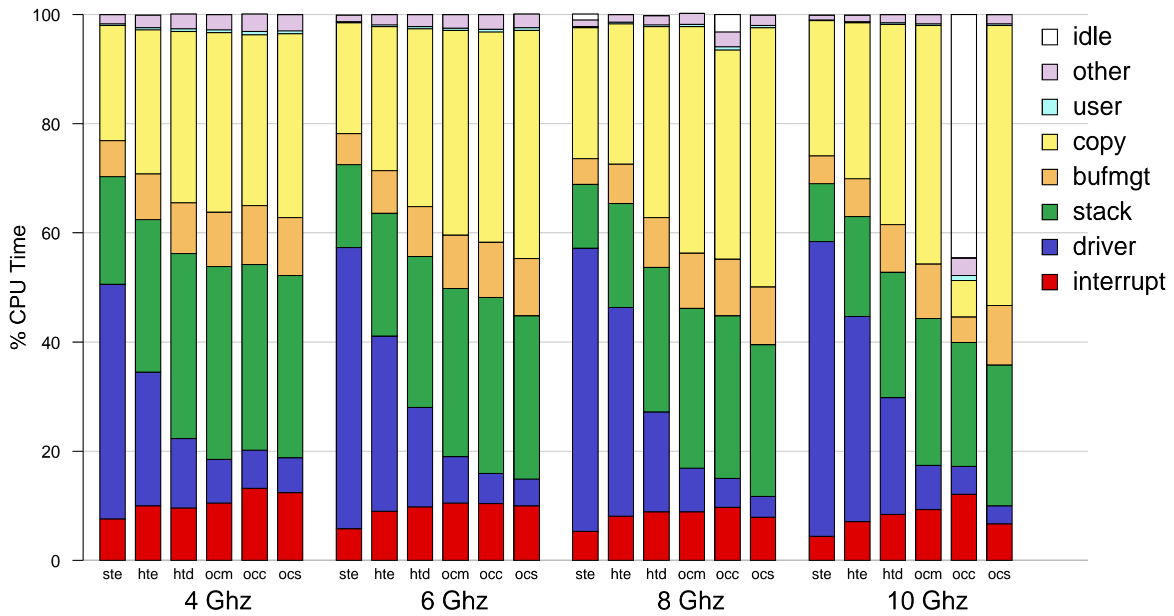

Figure 7: CPU utilization for the receive microbenchmark. (4MB L2 Cache)

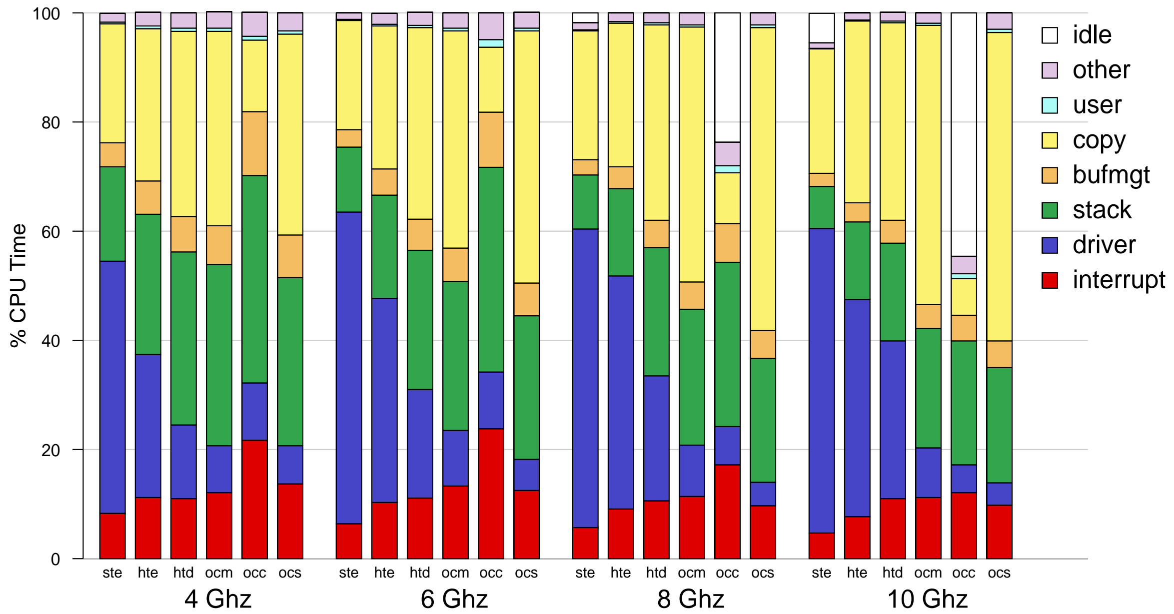

Figure 8: CPU utilization for the receive microbenchmark. (16MB L2 Cache)

Figure 9, Figure 10, and Figure 11 present the performance, cache activity, 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.

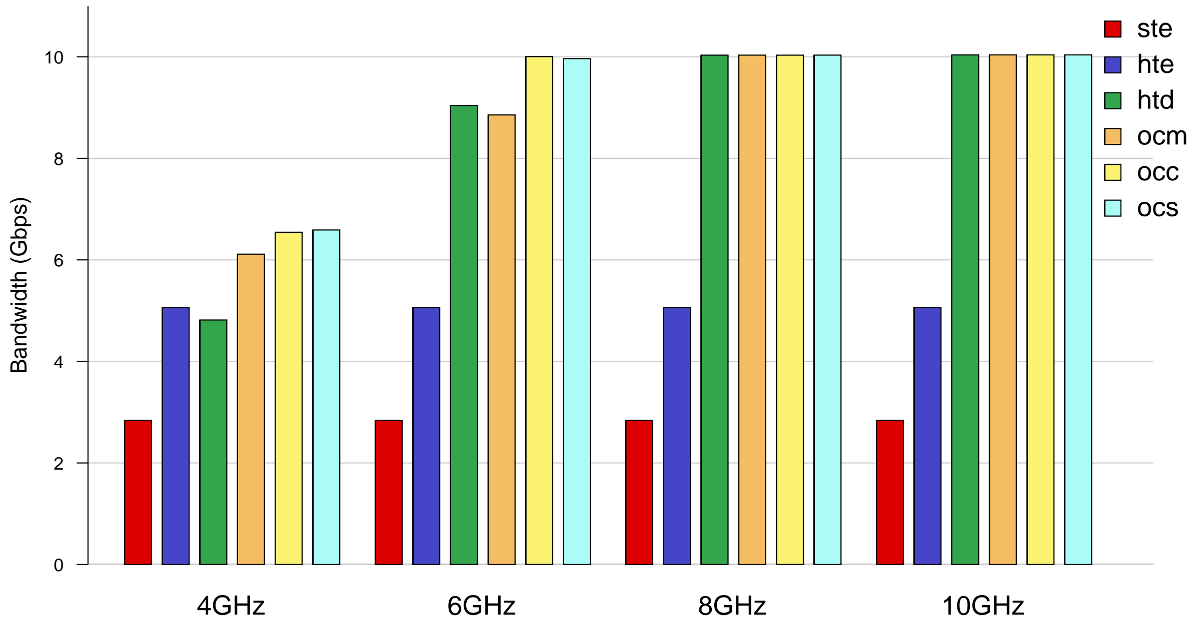

Figure 9: Achieved bandwidth for the transmit microbenchmark. (4MB L2 Cache)

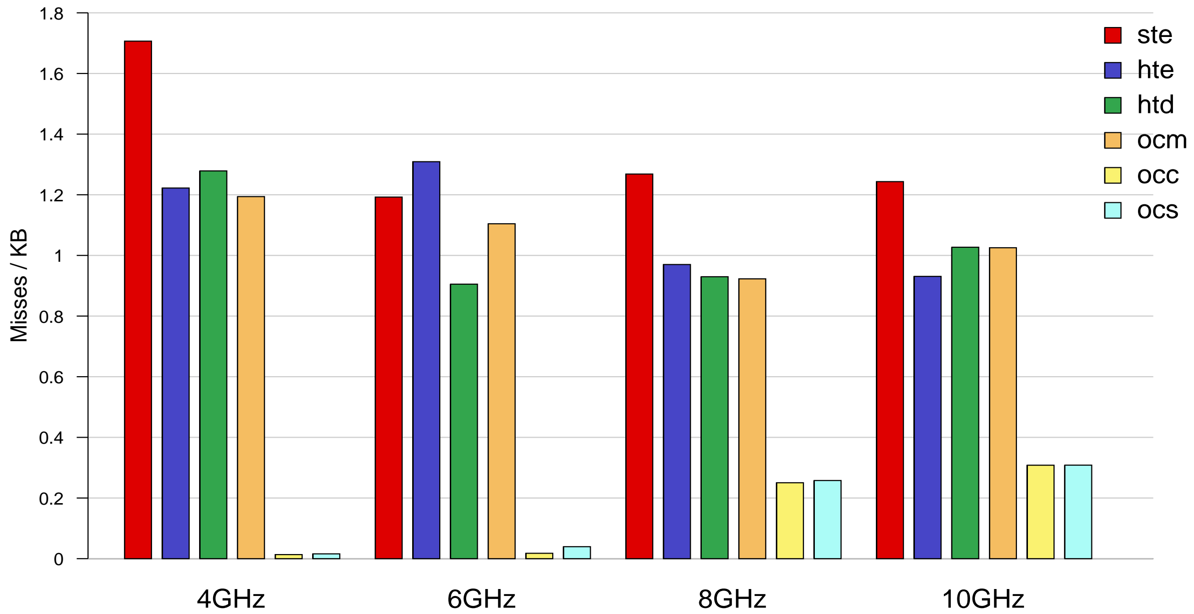

Figure 10: Misses per kilobyte for the transmit microbenchmark. (4MB L2 Cache)

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 DMA data

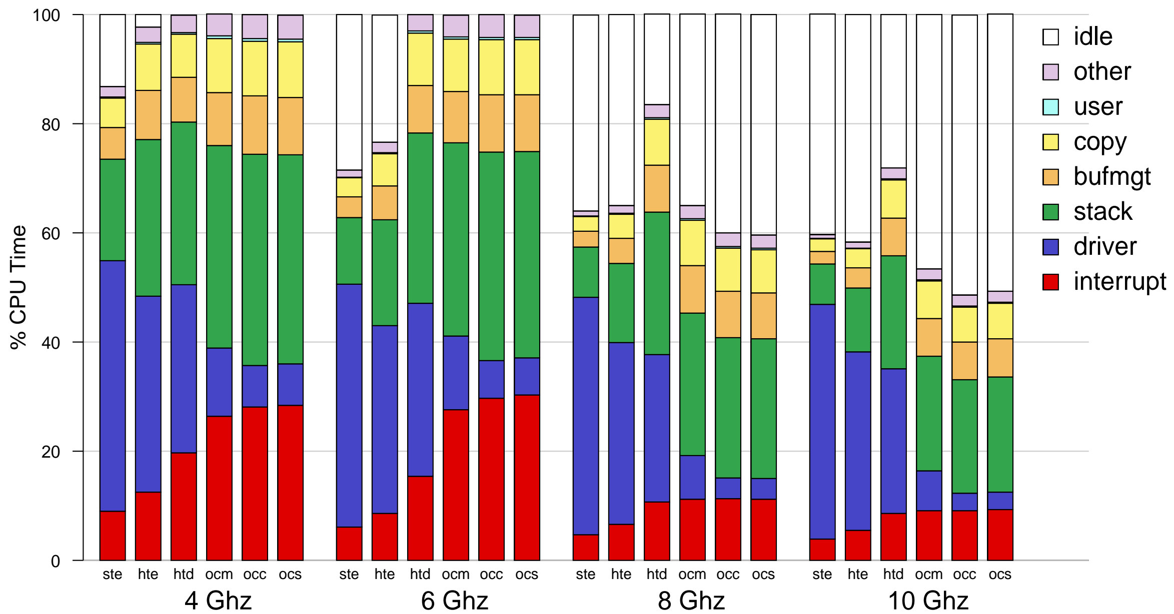

Figure 11: CPU utilization for the transmit microbenchmark. (4MB L2 Cache)

placement affects only headers and acknowledgment packets, giving OCC and OCS similar performance which is far better than 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 12, Figure 13, and Figure 14. 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 is unsurprising since the working set of this application is 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

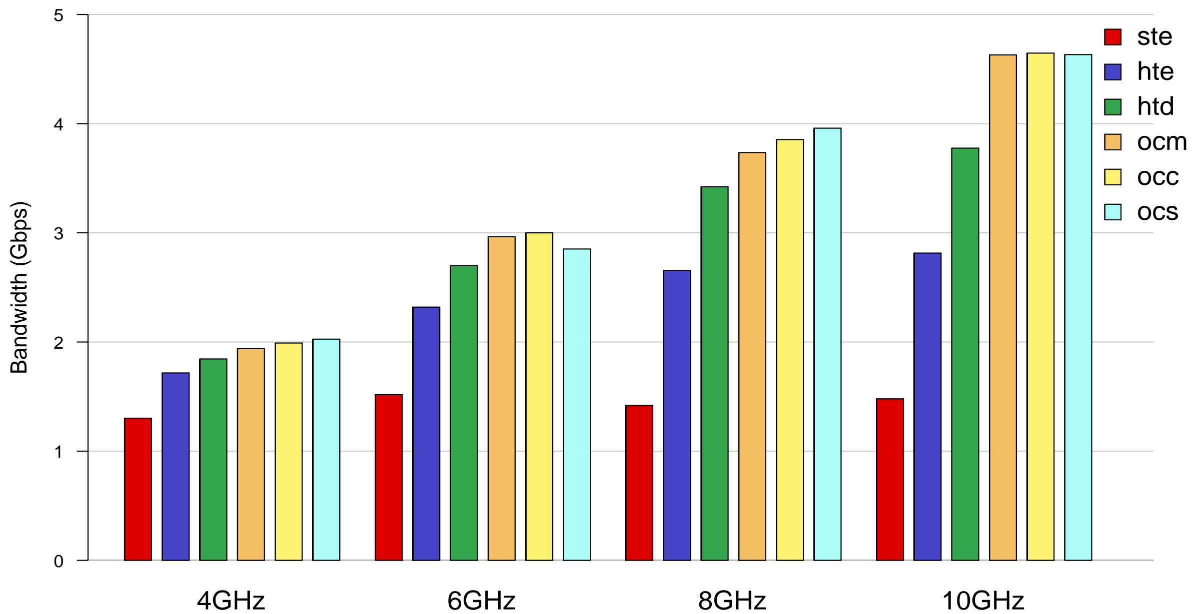

Figure 12: Achieved bandwidth for the web benchmark. (4MB L2 Cache)

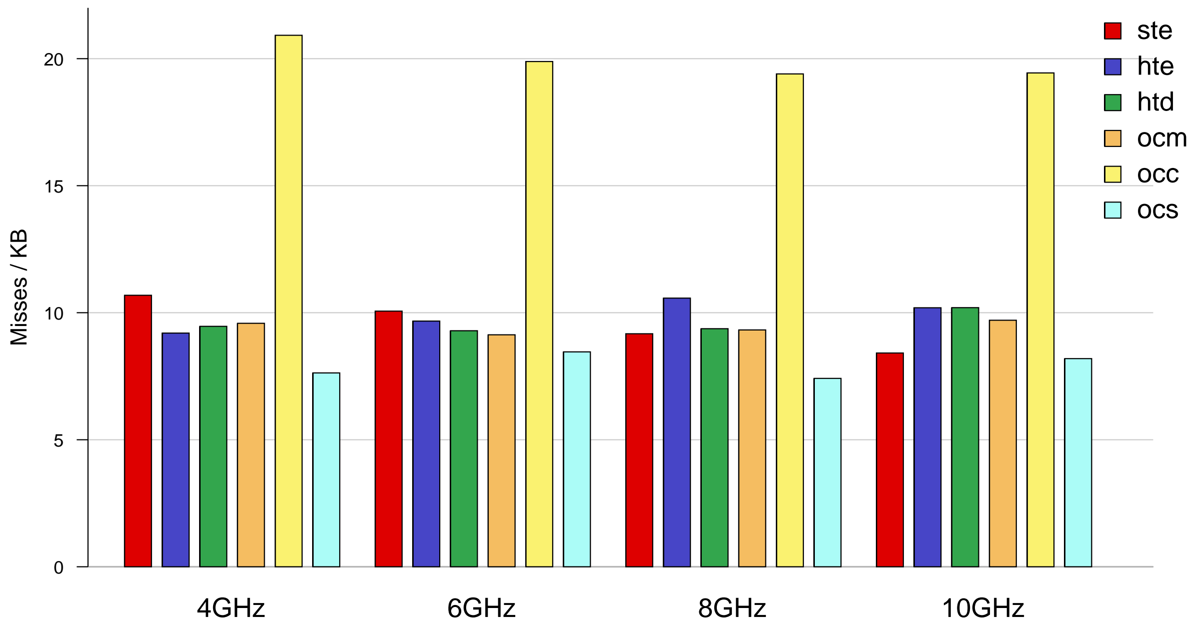

Figure 13: Misses per kilobyte for the web benchmark. (4MB L2 Cache)

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 15, Figure 16, and Figure 17 show 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

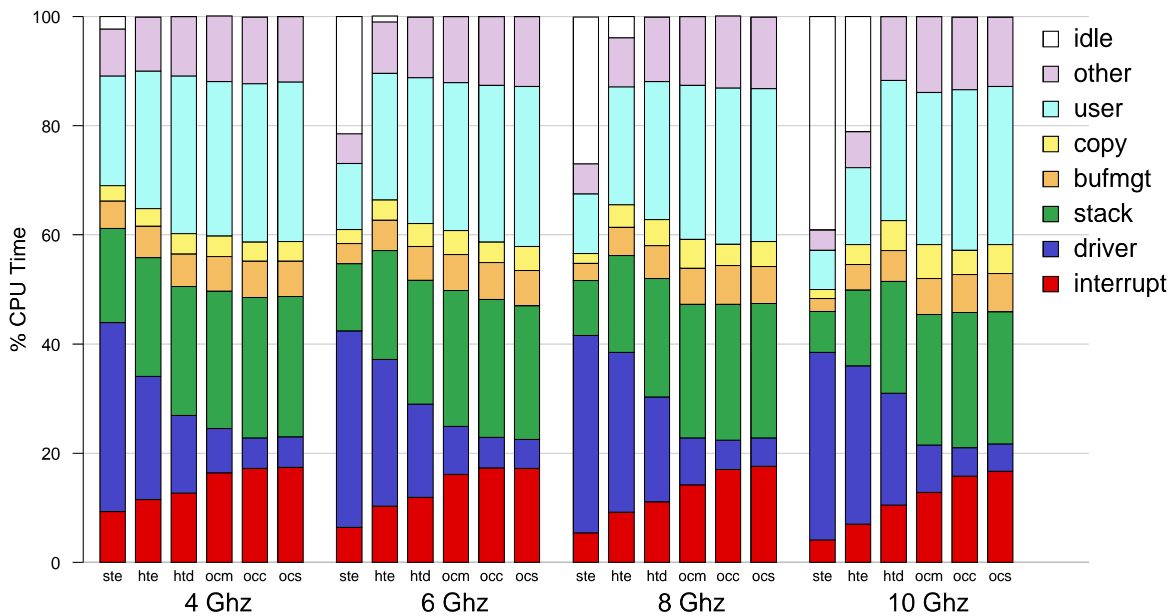

Figure 14: CPU utilization for the web benchmark. (4MB L2 Cache)

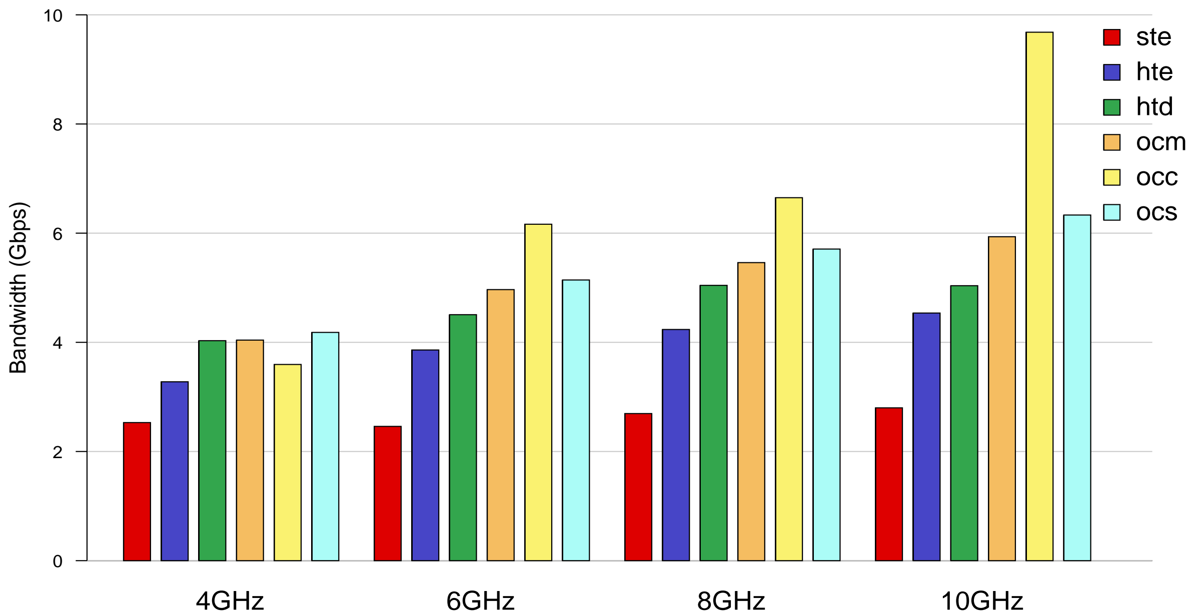

Figure 15: Achieved bandwidth for the NFS benchmark. (4MB L2 Cache)

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 resulting in the appearance of the working set size 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

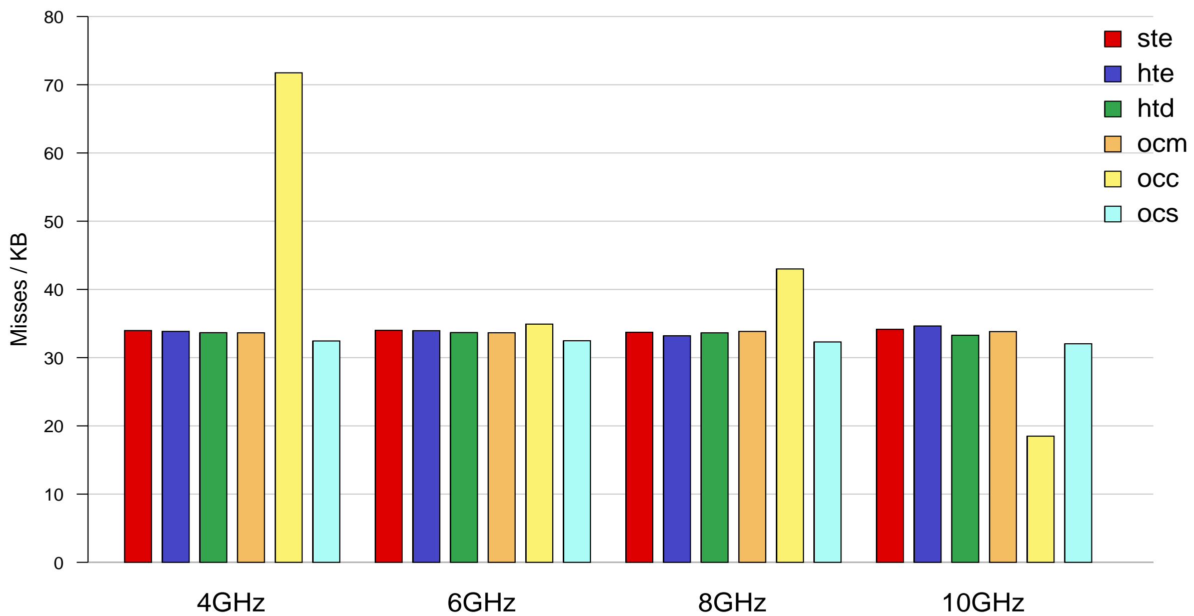

Figure 16: Misses per kilobyte for the NFS benchmark. (4MB L2 Cache)

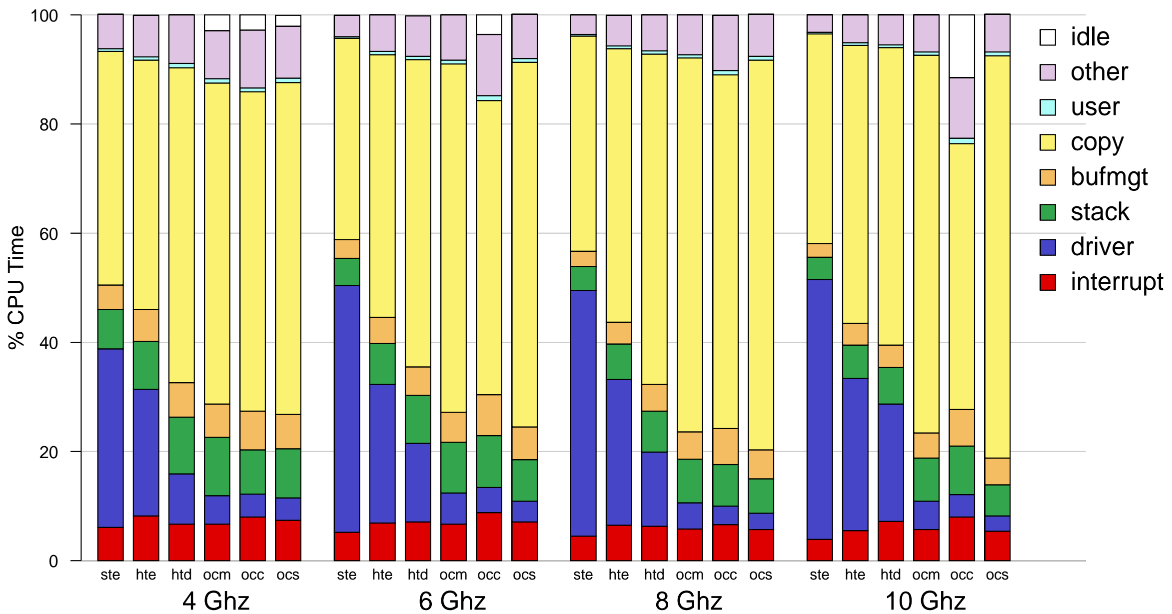

Figure 17: CPU utilization for the NFS benchmark. (4MB L2 Cache)

to the misses. In nearly all cases, these effects result in improved bandwidth due to the loosening of the CPU bottleneck.

Figure 18, Figure 19, and Figure 20 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 slower 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 poten-

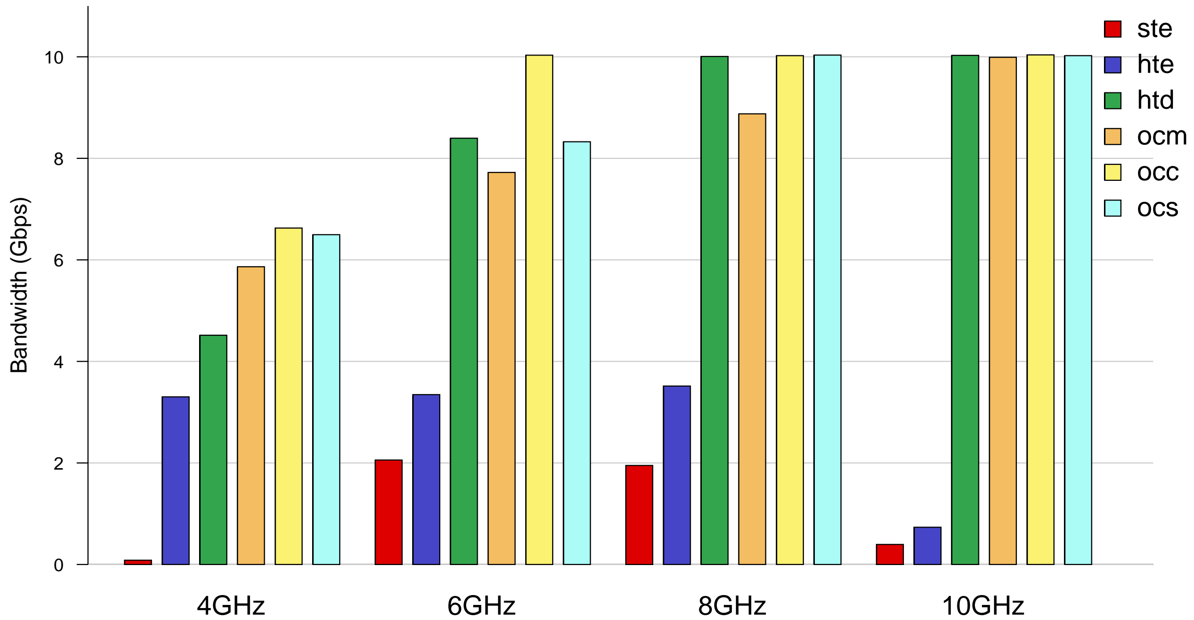

Figure 18: Achieved bandwidth for the NAT benchmark. (4MB L2 Cache)

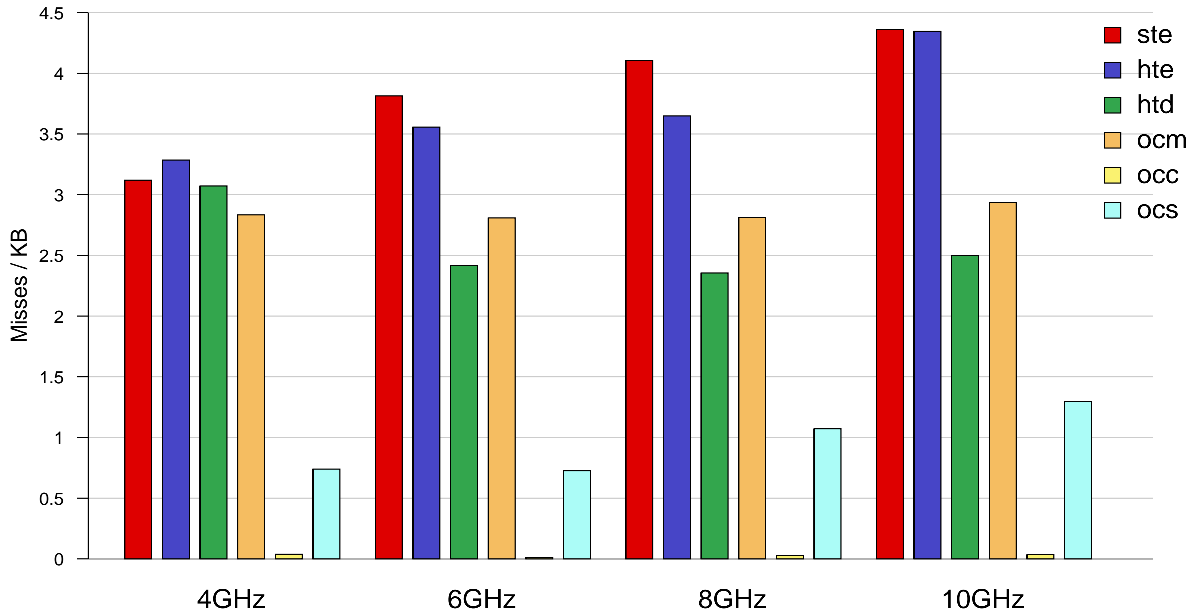

Figure 19: Misses per kilobyte for the NAT benchmark. (4MB L2 Cache)

tially a major cause of network congestion in our simulation. The configurations with unexpectedly significant idle time in their NAT runs have packets dropped at the NAT gateway, and we believe the poor performances are due to the TCP stack attempting congestion control at the endpoints. However, our runs do not run long enough to achieve steady state to confirm this, and we are still looking into other possibilities. Despite this, our hypothesis makes sense since the faster

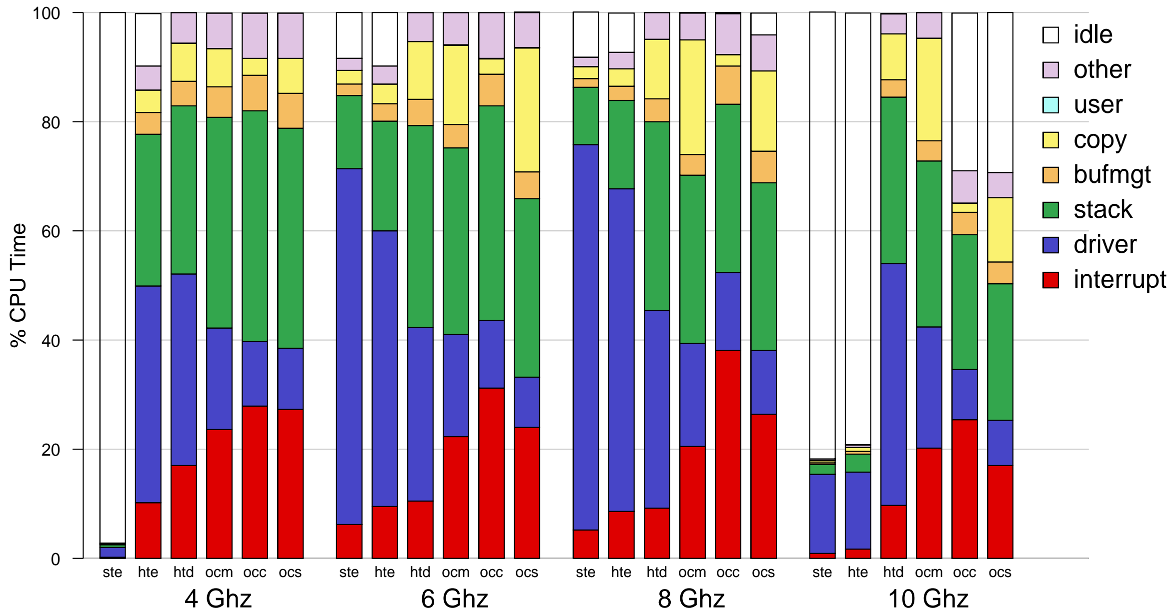

Figure 20: CPU utilization for the NAT benchmark. (4MB L2 Cache)

CPUs coupled with the low-latency NIC placements do not have trouble because 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 yield 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. 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 [12] 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 [18] and the J-Machine [8]. Mukherjee and Hill [17] 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 [11], 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. Reports indicate that the upcoming Sun Niagara processor, a multithreaded chip multiprocessor designed for network workloads and scheduled to ship in 2005, will have four integrated 1 Gbps Ethernet interfaces.

We have also done some previous work [1] showing that integration into the memory hierarchy has more potential to eliminate the bottleneck between the NIC and CPU than by merely improving bandwidth and latency between the two, which led us to our current work.

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 [6, 9] and user/kernel context switching [4, 10, 22]—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 [14, 23] would be unnecessary, as an integrated NIC would share the same high-bandwidth channel to main memory as the CPU.

Recent work from Intel's Communication Technology Lab [19, 19] shares our skepticism regarding offloading as an appropriate solution for high-speed TCP/IP processing. Their work on "TCP onloading" is more focused on system software architecture, and as such is highly complementary to our investigation of system hardware organizations. There is some overlap between our efforts; for example, their proposal for "direct cache access" for pushing NIC data into an on-chip cache from an off-chip NIC echoes our study in this paper. Their evaluation also focuses on prototyping more than simulation, allowing them to look at larger workloads but a more constrained set of hardware configurations.

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. 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 [24].

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 [8, 18].

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.

8. Acknowledgement

We would like to thank Kevin Lim and Jennifer Treichler for their help in preparing this report. 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.

9. References

[1] Nathan L. Binkert, Ronald G. Dreslinski, Erik G. Hallnor, Lisa R. Hsu, Steven E. Raasch, Andrew L. Schultz, and Steven K. Reinhardt. The performance potential of an integrated network interface. In *Proc. Advanced Networking and Communications Hardware Workshop*, June 2004.

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

[4] 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.

[5] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram A. Saletore, and Annie Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, February 2004.

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

[22] 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.

[23] 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.

[24] 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.