Integrated System Architectures for High-Performance Internet Servers

by

Nathan Lorenzo Binkert

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctorate of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2006

to Amity, my wonderful wife

ACKNOWLEDGEMENTS

Finishing this dissertation ends one major chapter of my life and marks the beginning of another. As with any major step in life, I could not have gotten through it alone and am incredibly fortunate to have had so many fantastic people help me along the way.

I first have to thank Amity, my best friend and wife. She was with me through the whole process of me getting a Ph.D., from beginning to end, always supportive, even when the end never seemed to be in sight. I can always count on her to be helpful and understanding when I need her to be. She lets me be who I am. I don't know many spouses who would have allowed, let alone supported, my decision to spend time and money learning to fly while in school. This is why I love her, she knows me. I don't know if I can ever properly express my gratitude and love for her.

I must also thank my parents, Peter and Jackie Binkert—they taught me to love learning which has led me to experience so many wonderful things. They have always been completely supportive of me and did whatever they could to help me learn and experience new things. My dad gets some extra thanks for actually reading and editing my dissertation, even when I needed it at the last minute.

I mentioned my wife and my parents first, but everyone in my family has helped me in some way. Kevin Binkert, my older brother, whose talents inspired me to excel, taught me that I had to choose where I wanted my life to go. Nonno, perhaps without even realizing it, taught me that a person's limitations are mostly self-imposed. I belive that a great deal of my creativity came from my nonna. Zia Ada is like another grandmother, always caring for me. Grandma Raye showed me, as she does everyone that she meets, that love is an awesome power that can get you through anything.

I must thank the many outstanding professors and researchers that helped me throughout this whole process, especially my advisor, Steve Reinhardt, Joel Emer, and Trevor Mudge. Steve has been a wonderful mentor, he gave me the freedom to explore my ideas and was always willing to work through them with me. He was patient through my myriad tangents as I would always find off-topic things to think about and do along the way. Those tangents usually led to incredibly interesting conversations about all sorts of things in computers and technology. Joel was another great mentor of mine and has my deepest respect as a researcher and person. No matter what the problem, Joel is always able to reason about it and provide new insight. As an intern, I could not have asked for a better supervisor than Joel. I could always count on Trev's quick wit to give me a laugh and his perspective when I couldn't see the forest for the trees.

I, of course, could never have gotten this far without the other students, past and present in my research group: Wei-Fen Lin, Steve Raasch, Erik Hallnor, Lisa Hsu, Ali Saidi, Ron Dreslinski, and Kevin Lim. We've all had many late nights together working in the face of a deadline. It was tough work, but I wouldn't have picked anyone else to do it with. In particular, Lisa and Ali, the two students I've worked with most closely. I worked with one of the two of them on just about all of the work that was done for my dissertation. For several years, they both put up with being a junior student to me. I hope they learned as much from me as I learned from them.

I certainly wouldn't be where I am today without the unconditional support of my closest friends, Brian Concannon, Brian Mc Mullin, Jason Hui, Jeremy Epple, and Kory Ketelhut. They've always been there for me no matter what. I know that if I ever need anything, I can count on them.

There are so many other people who helped me through my life that I regret that I cannot name them all. Teachers, friends, acquantinces, and strangers have all impacted me in ways that I may never know. I am thankful for those connections and know that I am better for them.

Finally, I'd like to thank my son. You haven't even been born, yet you've given me inspiration and motivation.

Contents

List of Figures

List of Tables

LIST OF APPENDICES

Chapter 1

Introduction

In the past decade, the role of computers in society has undergone a dramatic shift from stand-alone processing devices to multimedia communication portals. This shift has been facilitated by the use of standard protocols which enabled different vendors to independently develop software that would interoperate on the Internet. There are two protocols in particular that drive the Internet: the Transmission Control Protocol (TCP), which provides reliable, in-order delivery of data; and the Internet Protocol (IP), which facilitates the naming of Internet endpoints and delivery of messages to those endpoints. The explosive growth of the Internet moved TCP/IP networking 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 virtual private network endpoints, while IP-based storage area networks (SANs) are emerging for high-end servers. There are already a number of bandwidth-hungry applications that can readily make use of 10 Gigabit networking, such as cluster and grid computing systems and file servers on local area networks. Many emerging technologies such as network-attached storage (NAS), iSCSI SANs, telepresence-based remote conferencing, and video streaming will further increase demand for very high-speed network connections. With Gigabit Ethernet effectively the default for current desktop systems, 10 Gigabit Ethernet connections will be required to avoid contention at any local-area server shared by a reasonable number of these client desktop systems [31].

Though server systems with 10 Gigabit Ethernet are available today, those systems are able to achieve only a small fraction of the throughput their network connections are capable of. For the industry to deliver production computer systems capable of meeting the demands of high-speed networking, system architects must begin addressing this problem. This dissertation specifically investigates solutions to this problem by developing the simple integrated network interface controller (SINIC). SINIC improves the performance of networked systems by integrating the network interface controller (NIC) onto the CPU die. NIC integration alone can double the performance of a conventional peripheral NIC. Additionally, being on-die leads to a new set of NIC design parameters. These parameters are exploited to make SINIC a more flexible, yet less complex NIC design while achieving the same performance as a more complicated integrated NIC.

1.1 Networks are improving faster than processors

In recent years, network bandwidth is one of the few technologies that has outstripped Moore's Law. From 1995 to 2002, the IEEE Ethernet standard evolved from a top speed of 100 Mbps to 10 Gbps (10GigE), a hundred-fold improvement, while in the same period the 18-month doubling rate of Moore's Law indicates a mere 25x increase in transistor density (traditionally correlated with CPU performance). As a result, the host computer systems at the endpoints of these high-speed Ethernet connections are no longer able to keep up with the network data rate. Figure 1.1 depicts the relative increases of transistor density, network bandwidth, and I/O bandwidth. This graph shows that the rate of increase in network bandwidth is higher than the rate of increase in I/O bandwidth. This graph also indicates that Ethernet bandwidth is currently on the same order as I/O bandwidth. Additionally, transistor density does not increase as quickly as network bandwidth, making it necessary for network communication to be implemented more efficiently with a given number of transistors. These trends show that TCP/IP network I/O can clearly no longer be considered an afterthought in computer system design.

Despite the rapid increase in available network bandwidth, NICs in mainstream computers continue to be treated as generic peripheral devices connected through standardized I/O buses. This trend is partially historic as there is a long legacy of peripheral NIC devices. Additionally, mechanisms for improving the performance of such devices have not been obvious. The primary advances in end-host networking performance in the past several years have involved enhancing the NIC to allow it to offload simple tasks from the CPU such as direct memory access (DMA)[1], checksum generation, and segmentation of large transfers. Some portions of the industry are pushing this direction further by offloading much of the TCP protocol stack to the NIC. However, networking performance is a system-wide issue, involving the I/O

[1]to offload copies

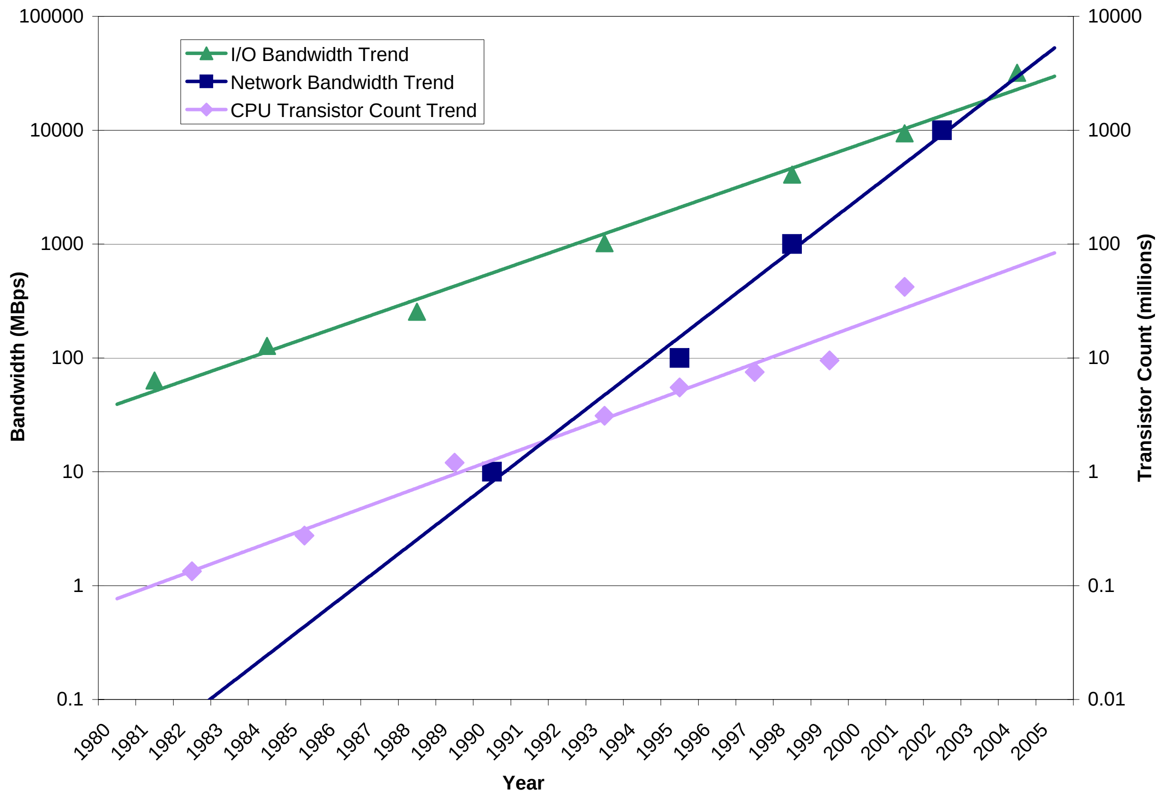


Figure 1.1: Performance Trends - The growth rates of various key system parameters plotted on a logarithmic scale, including: CPU transistor density (roughly correlated to CPU performance), I/O bandwidth, and network bandwidth.

subsystem, memory hierarchy, and CPUs. Attempting to address this system-wide problem with a point solution, as these TCP offload engines (TOEs) do, involves fundamental drawbacks, as will be discussed in Chapter 8.

Another call for a more central role of networking in computer system design is the fact that I/O standards evolve relatively slowly. This leads to situations where their performance can lag behind that of both CPUs and networks. As an example of the I/O standards lagging, the first few generations of 10 Gbps Ethernet (10GigE) cards, which were capable of 20 Gbps total bidirectional network throughput, were limited to a theoretical peak bandwidth of 8.5 Gbps by the 133 MHz 64-bit PCI-X I/O bus they used. Considering that the cards could be plugged into systems containing a 3.2 GHz CPU and an 800 MHz 64-bit memory bus, it's not difficult to see the discrepancy. As a stopgap, Intel dealt with the bandwidth gap by introducing a feature called the "Communication Streaming Architecture" (CSA) [42] on some of its chipsets. CSA was simply a dedicated, higher-performance I/O bus solely for the NIC's use. It failed to achieve a significant market share, partially due to the lack of new devices with a CSA interface and partially due to the advent of newer busses. These newer busses, PCI Express and PCI-X 2.0, have largely alleviated the bandwidth gap. Though these busses addressed the raw bandwidth mismatch, it was several years before 10GigE devices using newer busses arrived on the market. With even higher performance Ethernet (e.g. 40Gbps or 100Gbps) on the horizon, there is a significant chance that the bandwidth gap will return.

While I/O bandwidth is currently on par with network bandwidth, current I/O bus architectures fail to fundamentally reduce the latency of communication between the CPU and the network interface. This latency currently stands at thousands of CPU cycles (see Section 4.3) and directly affects the achievable bandwidth of the network interface (see Chapter 5) due to the nature of how system software controls the NIC and the length of this communication delay. Even though sufficient bandwidth might be available for communication between the CPU and the NIC, the latency often makes it difficult to fully utilize this potential. If this latency can be significantly reduced, the fundamental communication mechanisms between the NIC and the CPU can be redesigned to achieve higher efficiency. Increased efficiency is necessary to cope with the differences in scaling of the various system parameters shown in Figure 1.1.

The current industry approach to addressing network bottlenecks is to optimize the interface between the NIC and the CPU [13, 17, 21, 24, 84] without changing the system architecture. 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. As mentioned above, the most aggressive NICs available 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. The major drawback to the offload approach is that modifications to the CPU/NIC interface require specialized software support on the CPU side (see Section 8.1.4). The task of interfacing the operating system to a specific device such as a NIC normally falls to a device driver. 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 (see Section 2.3). From the software perspective, these NIC 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—a feature that is common today but unavailable just a few years ago—is useless unless the operating system has the capability of detecting this feature and disabling its own software checksum. The TOE approach of shifting significant amounts of protocol processing to the NIC requires an even more radical re-wiring of the kernel. This complicates and may even preclude deployment of future protocol optimizations. Finally, the problem of the intelligent NIC requiring modifications to the OS design goes both ways, thereby exacerbating the difficulties of implementing protocol offload. If an OS designer wants to support some common features such as packet filtering or tunneling, the NIC must also support these features or the OS has to bypass the protocol processing on the NIC and lose all of the benefits.

1.2 Integration of the NIC and the CPU

This dissertation takes a system-level view of high-bandwidth TCP/IP network performance. Using simulation, I analyze the performance of current and hypothetical future systems under network-intensive micro- and macro-benchmark workloads. I find that the key bottleneck in current systems is not solely protocol processing itself, but the high latency of communication between the CPU and the NIC. From a system-level perspective, this latency can be solved directly by moving the NIC closer to the CPU, either by placing it on a physically closer interconnect or by integrating it directly on die with the CPU. This integration does require area, but the requirements are modest and there is an abundance of transistors available.

Instead of trying to reduce the impact of high latency interaction, designers should do away with it all together by integrating the NIC and CPU on the same die. Once the CPU/NIC communication bottleneck is alleviated, protocol processing and memory copying become performance limiters. A non-integrated solution might rely on a dedicated TCP processing engine, whereas an integrated solution like the one proposed in this dissertation can exploit the increases in CPU resources that are becoming available in chip multiprocessor platforms.

Where this CPU speed increase is not adequate, it is often due to the CPU stalling while copying data. An on-chip NIC can access the last level of on-chip cache, allowing it to place incoming network data directly in the cache to address this copying overhead. This technique can also be used with off-chip NICs if the CPU cache allows data to be pushed in asynchronously from an external source [38]. Although some benchmarks occasionally see higher miss rates due to cache pollution from this technique, one does not observe any negative performance effects in the benchmarks. 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.

With the NIC and CPU integrated onto the same die the properties of the communication between the NIC and CPU are completely changed. Bandwidth is increased while latency is decreased, both potentially by several orders of magnitude. Given these drastic changes, it is reasonable to conclude that the traditional NIC should be redesigned to take advantage of these properties. The latency-hiding mechanisms of modern NICs, scatter-gather I/O and NIC-driven DMA , may no longer be necessary. Getting rid of these two mechanisms can reduce the complexity of the NIC and increase system flexibility. In addition, acceleration hardware such as checksum generation can be controlled by the host CPU and can potentially benefit other applications.

Although, on the face of it, an integrated NIC seems like it might provide less flexibility than an add-in card, there are reasons why this need not be true. In fact, tighter integration can lead to a significantly more flexible system since the NIC can rely on a normal CPU to do its processing, allowing the operating system designer to explicitly manage the way the NIC works.

In terms of flexibility, the dominance of Ethernet makes the choice of link-layer protocol almost a non-issue, while simple off-chip physical interface devices could allow selection of different media. In addition, a simple flag on the CPU—with some cooperation of the system software—could even disable link layer framing and pass the transport layer packets out of the CPU for framing by an off-chip interface. While Ethernet may be the default choice supported by the CPU and a single 10 Gbps connection should be more than adequate for a significant fraction of the market, there is no reason to bind the speed of the interface to a specific protocol implementation like 10GigE. Instead, the NIC speed could be varied as memory bus speeds are varied. As the CPU speed increases, the network interface speed could increase as well. This non-standard speed can easily be managed using off-chip switching to convert the traffic into multiple slower connections as necessary. Single-chip multiprocessor server devices could sport different speed off-chip interfaces as a product differentiator. High-end multi-chip systems can share integrated NICs across their inter-chip interconnect in much the same way that modern systems based on AMD Opteron processors exploit their HyperTransport channels to share DRAM and I/O devices [1]. High-end cluster-based systems could even eliminate standard link-layer protocols, and use a high-speed, low latency proprietary interface [41].

1.3 The Challenge of Evaluation

Evaluation is a key challenge when investigating alternative network system architectures. Analyzing the behavior of network-intensive workloads is not a simple task, since 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 [21] or emulate them using programmable NICs [13, 17, 24, 84, 47]. Unfortunately, this approach is not feasible for my proposal because I am not simply modifying the NIC but rather integrating it onto the processor die itself.

To cope with these difficulties, I have developed a simulation environment for the M5 simulator [7] specifically targeting networked systems. The development of this system represents a significant amount of work, several man–years, and it is, as a result, unique in its ability. No other simulator, academic or commercial, supports a detailed network I/O model capable of providing realistic timing measurements. M5 is capable of simulating server and client systems along with a simple network in a single process, enabling it to provide deterministic, repeatable results. M5 uses full-system simulation to capture the execution of both application and OS code and models the hardware platform with enough fidelity that it can boot a variety of operating systems, including Linux and FreeBSD, without modification. In addition its I/O and full-system capabilities M5 includes a detailed out-of-order CPU, an event-driven memory hierarchy, and a detailed Ethernet interface device. Finally, in order to promote further research in network architecture, this system is publicly available [52].

1.4 Thesis Statement

The traditional treatment of NICs as standard peripheral I/O devices makes it difficult for modern systems to take advantage of the bandwidth available on modern networks. This dissertation demonstrates the importance of tighter integration between the NIC and CPU. Tighter integration alone provides significant benefits, but also enables a redesign of the NIC itself to take advantage of the new properties of the interactions between the NIC and CPU, particularly lower latency. A suitably redesigned NIC enables software optimizations not possible with traditional NIC designs.

1.5 Contributions

The major contributions of this dissertation include:

- the development of a simulation environment that is specifically targeted for networked systems;
- the analysis of several networking benchmarks and the understanding of specific characteristics of networking workloads and how they behave when using simulation and associated techniques;
- the finding that simply integrating a traditional NIC onto the CPU die can lead to performance improvements of more than 2x over conventional peripheral based designs;
- the determination that cache placement of DMA data has the potential for tremendous wins in performance in some cases while not affecting performance and increasing cache miss rate in others;
- the development of a new network interface controller design that is optimized for the low-latency interaction provided by integration; and

- the demonstration of a software optimization, zero-copy receive, that is made possible by this new NIC design.

1.6 Organization

Chapter 2 provides some background information on how networking workloads behave and where computing resources are spent. Chapter 3 describes the simulation infrastructure that was used to carry out the studies that follow. It details the organization of the M5 simulator, the primary tool used in these studies, and describes the aspects of this tool that make it especially suited for examining networking workloads. Next, Chapter 4 discusses the analysis of network workloads and describes the benchmarks and the methodology used by the studies performed in this dissertation. In addition, some general insights on the characteristics of networking workloads and the difficulties involved in analyzing the performance of these workloads are investigated there.

The first major study of this dissertation is presented in Chapter 5, which 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 (e.g. creating a TCP offload engine), this approach does not require significant software changes, making it possible to re-use existing device drivers with at most minor modifications. In this study, I compare several basic system configurations connected to a 10 Gbps Ethernet link. Several of these configurations model conventional systems with PCI Express-like I/O channels, while others model systems with different levels of tighter integration. In addition to investigating the network performance achieved using the different configurations, the effect of these configurations on parameters such as cache performance and CPU utilization is studied. Finally, Chapter 5 contains a preliminary study on how placement of NIC data in the cache vs placement in main memory can affect the system.

The second major study begins in Chapter 6 of this dissertation. In this part, I assume an integrated NIC as a base design and investigate the design space of a NIC given that the fundamental bandwidth and latency parameters between the NIC and CPU have been changed by orders of magnitude. The design investigated in detail is the simple integrated NIC (SINIC). The SINIC design takes a conventional NIC and removes NIC-driven DMA and scatter-gather I/O capabilities. Because of this simplification, the normal functions of the NIC become the responsibility of the CPU and system software. This design increases the overall flexibility of the system and allows system software to implement more radical optimizations not available to traditional designs. This study continues in Chapter 7 where I introduce V-SINIC, an extended version of SINIC that provides virtual per-packet registers, enabling packet-level parallel processing while maintaining a FIFO model. V-SINIC also enables deferring the copy of the packet payload on receive, which I exploit to implement a zero-copy receive optimization in the Linux 2.6 kernel.

The studies done in this dissertation are followed by a discussion of other work relevant to this dissertation in Chapter 8. The related work includes comparisons with other hardware and software techniques for improving networking performance. Finally, in Chapter 9 I discuss future work and in Chapter 10 present my overall conclusions.

Chapter 2

Background

The TCP/IP protocol suite has driven the Internet for nearly 25 years. When the Internet protocols were initially conceived, network bandwidth was a critical resource. Also, at that time, the latency to main memory and the cost of a branch misprediction were not the the major concerns that they are today. This led to protocol designs that maximized the efficient usage of the network but were not optimized for the types of processors that we have today. Since the protocols were developed, network bandwidth has become plentiful, the gap between the processor and memory began to form and widen, and control speculation due to superscalar and out-of-order processing became a concern. Throughout this time, a guiding principle in the journey to higher bandwidth has been a well known rule of thumb: roughly 1 Hz of CPU speed is required to achieve 1 bps of bandwidth. Unfortunately, given that CPU clock speeds are not expected to increase at as dramatic a rate as they have in the past [82], it will become necessary to find other means to improve the performance of end-host networking. As seen in [30], the 1bps/Hz ratio while roughly correct does not track the actual frequency changes of modern processors. In fact, the ratio appears to be decreasing over time, despite the fact that CPUs are doing more computation per processor clock cycle (the Hz component). These developments drive a great deal of researchers to propose newer, better protocols, yet the TCP/IP protocol suite has endured largely unchanged due to the vast installed base of software. While there are still proposals for newer protocols that lend themselves to improved performance, a great deal of research investigates methods for allowing systems to deal with the inefficiencies of these venerable protocols without changing them.

There are several protocols at the core of TCP/IP protocol suite. First and most important of these is the Internet Protocol (IP) [68]. IP is a network protocol whose main task is to provide the information that the routers need to identify the source and destination hosts of a packet. On top of IP are two transport layer protocols, the Transmission Control Protocol (TCP) [69], and the User Datagram Protocol (UDP) [67]. TCP is a complex protocol that provides reliable, in-order delivery of packets as well as data integrity. UDP, on the other hand, is a very simple protocol that provides no delivery guarantees beyond data integrity—packets can be duplicated, dropped, or reordered, and it is up to the user of UDP to determine the best matter to deal with this.

This chapter provides some background on TCP/IP processing, specifically addressing the ways that the operating system and hardware interact and how the hardware performs during each step of the process. The information presented here primarily focuses on the approaches and issues with the existing TCP/IP protocols and does not investigate performance improvements that require modification of the protocols. A review of the body of work in improving these protocols and their interaction with the hardware can be found in Chapter 8.

2.1 The System Call Interface

Before diving into a discussion of the steps in TCP/IP processing, it is necessary to examine the semantics of the user system call interface and the way that kernel buffers are managed because they have a major impact on performance. The standard network socket protocol defines that the interface uses copy semantics [69]. For a write, the subsystem handling the user's write must make a copy of the data if it wants to use data after the write returns. For TCP, this copy always happens because the data must be preserved in case a retransmission is required. For a read, the user provides a buffer in which the socket layer should place the data. Because the socket layer lives in the operating system kernel, and the user of the socket layer is user-space code, these copies are commonly referred to as user-to-kernel or kernel-to-user copies. It has been shown that these copy operations represent a significant percentage of the total overhead at the end host [18]. Work on avoiding this copy, at least for received packets, represents a large fraction of the research in improving system performance (see Chapter 8.3 for details).

2.2 Kernel buffering

Since these user-to-kernel and kernel-to-user copies alone are so expensive, the kernel goes to great lengths to avoid other copies internally. These copies are avoided by

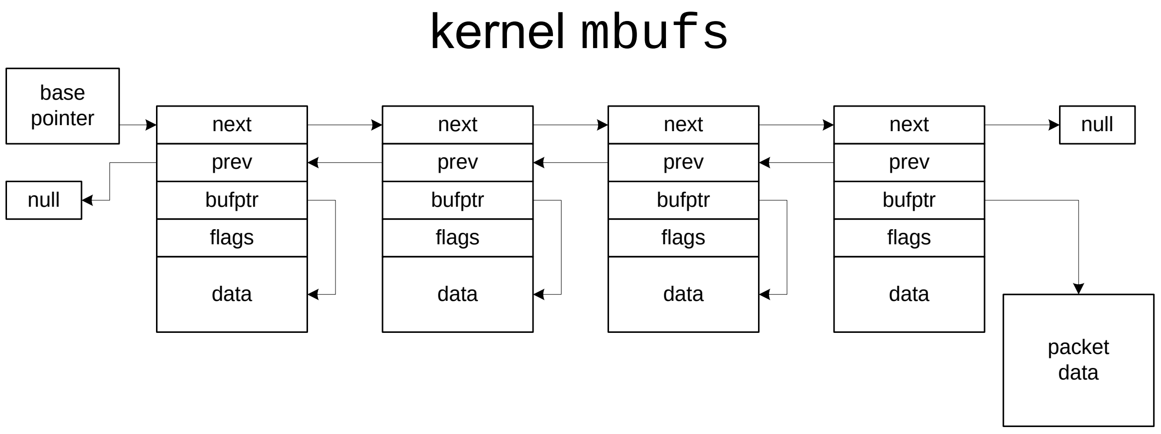


Figure 2.1: Kernel `mbufs` - The kernel data structure used in BSD derived operating systems for storing network data. The helper functions provided along with the `mbuf` data structure are designed to help the kernel avoid copying data.

employing sophisticated buffer management techniques. Universally, the key abilities necessary for these buffers are the ability to prepend data to a packet as a transmitted packet gets built, and to remove data from the front of a packet as protocol layers strip off headers before passing the packet to the next level.

To avoid copying, the kernel keeps lists of objects that hold information about buffers. Since the original BSD networking stack used a data structures called `mbufs`, and they are commonly used throughout the literature (and continue to be used today in BSD derived operating systems), examples here will use `mbufs`. In Linux, the `sk_buff` structure plays the same role and for the purposes of the discussion here can be considered equivalent. Each `mbuf` (see Figure 2.1) has among other things, a pointer to the start of the data, a data length field, next and previous `mbuf` pointers, a flags field, and a small data buffer. For large buffers, the `mbuf` uses an external data buffer, and has additional fields: a pointer to an external data buffer, the size of the external data buffer, a list of users of that external buffer for reference counting, and a function pointer for freeing the external buffer once there are no more referents.

The linked list design along with the adjustable pointer to the data allows the kernel to add data to and remove data from either end of a packet. The vast majority of network interfaces support scatter/gather DMA (see Figure 2.3 and use lists of buffers in a similar way, but unfortunately, the lists use different formats, and the kernel must translate data from one format to the other as data is sent to the network, and as buffers are set up to receive packets.

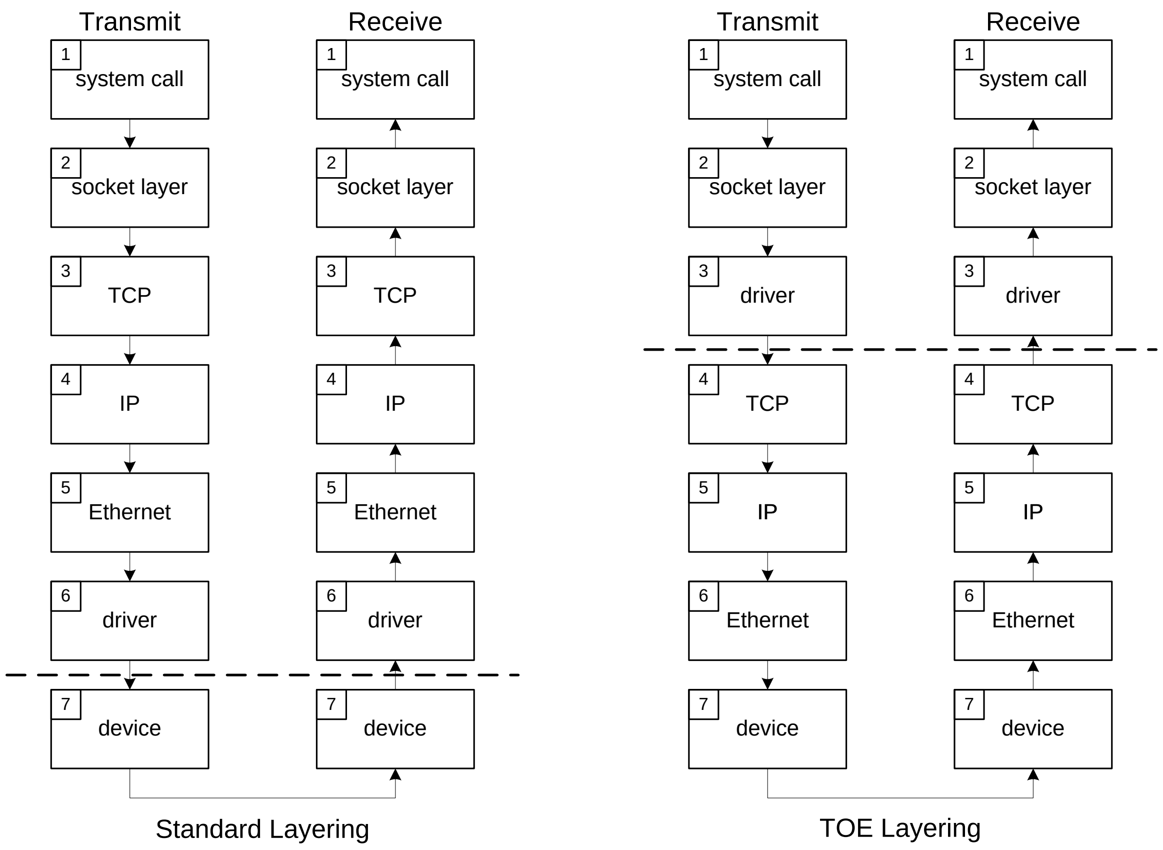


Figure 2.2: TCP/IP Processing Steps - The various steps of TCP/IP processing.

2.3 Overview of TCP/IP Processing

This section provides an overview of the transmission and reception of TCP/IP packets from both a hardware and a software point of view. On the transmit side, it begins with a user process on one system calling the `write()` or `send()` system call traverses through the kernel, the device, and the link. Continuing on the receive side: the link, device, kernel, and response to the user process. The overview details where the data is going, what the software is doing, and how it affects the hardware. It begins with a discussion on how buffering is done in the kernel, and follows with a discussion of the transmit and receive paths of the network data flow. This overview avoids discussing a specific implementation and instead present an overview of the flow. The flow follows Figure 2.2, which shows the high level block diagram of a a sender and receiver using TCP, IP, and Ethernet. For a more thorough discussion, see [15].

2.3.1 TCP/IP Transmit Path

System Call

Once a user process decides that it will transmit a packet, it must signal the kernel that it wishes to do so with a system call. The system call is in some ways like a normal function call, except that in addition to jumping to a new place in memory, the processor switches to a higher privilege mode (commonly referred to as kernel mode). Often, once the processor has jumped into kernel code, many of the instruction references will miss in the cache because it has been long enough since the system call handler has been invoked that the instructions have been pushed out of the cache. The transition from user-space into the kernel typically requires a pipeline flush because the kernel privilege is often a global state allowing instructions of only one privilege to exist in a machine at a time. Once inside the kernel, the system call handler code must copy the data that the user wishes to transmit from the user-space buffer to a kernel buffer. These blocks of data may, or may not, hit in the cache depending on the application.

Socket

Once the data has been copied into kernel buffers, the kernel must then decide which connection the data to be transmitted belongs on and call the transport protocol's output function.

TCP

Assuming that the connection is TCP, the data must then be passed to the output function which determines the state of the connection. At this point in the processing, the data can continue to be processed and sent out the NIC or it can be queued for later transmission. Transmission can be delayed for a few reasons, including delaying small sends according to Nagle's algorithm [60] to combine them into larger packets and delay due to congestion control based on TCP's sliding window algorithm. If the data is delayed due to either Nagle's algorithm or congestion, the data is queued for transmit at a later time. If the system is ready for transmit, then `tcp_output()` will then build the proper TCP header and prepend the header to the packet. In building the TCP header a checksum of the entire TCP header and payload must be computed. Finally, the TCP code must determine what the next protocol to call in the stack is. In the vast majority of cases, this will result in a call to the IP output processing function.

IP

The IP portion of the protocol handling code must fragment the packets, if necessary, build and prepend an IP header, and determine which interface to send the packet to. In the common case, TCP packets will not need fragmentation since TCP itself will divide up the data into appropriately sized segments based on the maximum transmission unit (MTU) of the link. The IP header contains information on the fragmentation and destination IP address and also includes a header checksum. The next step of protocol processing involves looking up the destination IP address in the routing table to determine the destination interface. The destination will in many cases be an Ethernet device, but can also be an encapsulation protocol such as IPsec [46], IP-in-IP [79], or PPP [78].

Ethernet

The common case for the link layer is Ethernet. The first step in the Ethernet protocol processing is in the device independent layer. Here, the destination and source Ethernet addresses are combined to form the Ethernet header and prepended to the packet. If the destination Ethernet address is not known, an address resolution protocol (ARP) [66] request must be constructed and sent out on the network to determine the correct destination address for the given IP address. Once the Ethernet

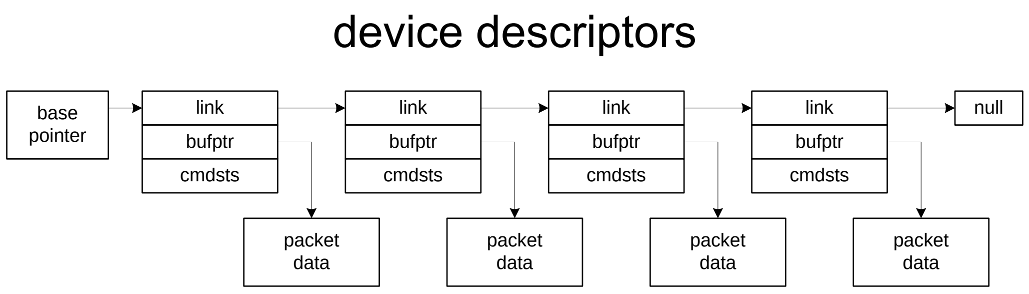


Figure 2.3: Device Descriptors - NICs that support scatter gather DMA generally have a descriptor chain that lives in main memory in order allow the device to DMA those descriptors. Descriptors, like `mbufs` (Figure 2.1), have control information and pointers to buffers. Generally, any memory that a descriptor points to is also pointed to by an `mbuf`. For example, a large fraction of the time spent in a transmit routine of a device driver is spent looping through the `mbufs` in a packet and copying the relevant information from the `mbuf` header to the device descriptor.

header is constructed, the packet is passed to the device dependent portion of the Ethernet layer.

Device Driver

It is the job of a device driver is to provide a software layer to allow the operating system kernel to talk to the device. For a NIC, the device driver is mainly converting from the kernel's buffer management scheme to the NIC's. The similarities between these schemes can be seen by comparing Figure 2.3, which shows typical device descriptors, and Figure 2.1, which shows kernel buffers.

The computation involved in the device dependent layer can vary greatly depending on the device itself, but in general, the device driver will find some available transmit descriptors to map the packet data to and signal to the device that there are new packets ready for transmit. The descriptors themselves are usually in a region of DMA memory that is mapped to the device. A descriptor usually includes a pointer to a physical address, a length, a pointer to the next descriptor in a chain of descriptors, and some flags. The last thing to touch the descriptor data is the device itself, so the write to the descriptor will miss in the cache. In many devices, the descriptors are not kept in a linear list, but rather in a linked list of descriptors in a small 1kB–4kB region of memory. If the descriptors are sufficiently large there could be a potential of multiple cache misses for each descriptor access. The linked list design is used for adapters that support prioritized queuing, though I've seen many drivers ignore this design and arrange the descriptors in a linear list. Once the descriptors are set up, the driver must indicate to the device that there is new data ready for transmit. This is generally done with an uncached store to the device's memory-mapped register space.

Device

At this point, the CPU's involvement with the transmission of the data is complete. Once the device receives the indication that there is more data to transmit and the device has determined that it has enough buffer space to buffer the packet, it first does a DMA read of the descriptors to determine what data is needed for transmit. That data was last touched by the CPU so it will have to provide the data or cause a coherence operation, then the read will complete. Next, the actual packet data is read from memory via another DMA read. Again, the CPU was the last thing to touch it, so a coherence operation is likely to take place to cause the CPU to write back its data before the read takes place. Finally, the NIC transmits the data on the wire. At this point, the NIC interrupts the CPU to indicate that the transmit descriptors are now free for the CPU to reuse.

2.3.2 TCP/IP Receive Path

There are seven basic steps involved from the time that the packet is received on the network interface until it is presented to the user process.

Device

The first major step, once the complete packet has arrived, is that the NIC uses DMA to transfer the packet into system memory. In order to do that, the NIC must know where to transfer the data. This information is usually stored in buffer descriptors (see Figure 2.3), which also must retrieved with a DMA. Once the packet has been completely copied into memory, the NIC sends an interrupt signal to the processor. This interrupt is a relatively long-latency operation because it must travel from the NIC through an IO bridge, and the memory controller before reaching the CPU.

Device Driver

Once the interrupt has arrived at the processor, the CPU must interrupt the currently running process and vector into interrupt handling code. If the interrupt line is not shared among multiple devices, this process can be very quick because the CPU can jump to the correct interrupt handler based on the interrupt that was signaled. However, PCI—the most common bus interface for modern NICs—has the ability to share interrupts, so it may be necessary to poll multiple devices to determine which one interrupted. This polling can either be beneficial, in the case where other devices do have pending work because you can service multiple events with one disruption of program flow, or detrimental, for the case where there are no other outstanding events. In the latter case, the CPU is wasting time doing extra device register accesses to query the other state of idle devices. The latency of a PCI device register access on a similar machine is approximately 3000 cycles. This lengthy delay will result in the processor stalling for the majority of that time since modern processors are unable to overlap such delays.

In addition to the costly device register accesses that must occur as a result of the interrupt, there are likely to be a large number of cache misses due to the fact that the interrupt handler code may not have been run in some time. This will of course also add to the total time taken to service an interrupt.

Once the interrupt handler has been invoked, the processor has to check the DMA descriptor list to determine what data is available. Since the device was the last thing to access the descriptor list, the descriptors will miss in the cache. On most modern devices multiple descriptors can fit within a single cache 64-byte block, so it is likely that there will only be one or two blocks that will miss. Once the descriptor is read, then the interrupt handler knows what data has been transferred, and can put a pointer to that data on the Ethernet receive queue.

Ethernet

The protocol stack takes the packet off the receive queue, and must then read the first byte of the packet to determine what must be done with it. Again, the device was the last thing to access this memory, and the processor will incur a cache miss. Most of the protocol processing within the kernel will happen on the first cache block worth of data, so there will be only a single cache miss to wait for. The protocol headers will fit within a single cache block provided that there are few options used. The total length of the headers is 54 bytes: 14 bytes in the Ethernet header, 20 bytes in the IP header, and 20 bytes in the TCP header. In most implementations, in order to properly align the IP and TCP headers, an extra two bytes of padding are appended to the front of the packet.[1] Within the protocol stack, the kernel must first determine if the packet was destined for the local host and if not, drop it. It must then look up the protocol type to check if it is an IP packet. When this is determined, the packet is then passed to the IP input processing function.

IP

In this function, the kernel first must verify the header checksum. Then if the packet is a fragment, the kernel must determine if the current packet completes a fragmented packet. If it does, the packet will be reassembled; if not, the packet will be saved to be reassembled later. The IP header of the packet (recently reassembled or not) is then examined to determine the IP protocol that should handle the packet. The most common decisions here are UDP and TCP. I'll assume TCP since it is the most common. The IP input function will then pass the packet to the TCP input function.

TCP

The TCP input function has to determine if the packet is correct by doing a checksum calculation. Once correctness has been determined, it then must check the packet ordering. If the packet was received out of order, it is put on a queue to wait for earlier packets to arrive. If the packet has been received in order, it is appended to the socket receive queue and, in many cases, a software interrupt will be scheduled to trigger the processing of the data on the queue.

Socket

When there is data in an application's received socket queue, a blocked application must be marked runnable or an application performing asynchronous I/O must be signaled.

[1] CPUs generally require that objects of particular sizes reside at particular offsets within memory. For example, most processors will require 32-bit integers to be stored on memory addresses divisible by four. The 32-bit values in the IP and TCP headers are offset at 4-byte boundaries from the beginning of their respective headers. If the 14 byte Ethernet header begins at a 4-byte boundary, this will result in all of the 32-bit integers in the IP and TCP headers being at 2-byte, but not 4-byte boundaries requiring unaligned accesses to retrieve those values. These unaligned accesses if handled in hardware are typically slower, and if handled in software generally require extra copying making them much slower.

System Call

If the application has been waiting for data with a blocking `read()`, or some other wait function, it will be marked ready. If the operation is a read the application then consumes some or all of the data, depending on how much was requested. If the operation was another type of wait, such as a `select()`, it must call `read()`. Since the data is in a kernel buffer, and the read system call requires the user to specify a buffer in which to place the data, the data must be moved from the kernel buffer to the user buffer. In most cases, this data movement requires all of the data to be copied from the kernel buffer to the user buffer. This copy is expensive. In addition to needing to copy a large chunk of data, one word at a time, it is likely that every block copied will miss in the cache because, as before, the adapter was the last thing to touch the data, and the kernel hasn't touched the data until now.

Chapter 3

Network Simulation with the M5 Simulator

Evaluation is a key challenge in investigating alternative system architectures for networking workloads. Unfortunately, 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 resort to prototyping them in hardware [21] or using programmable NICs [24, 17, 84, 13, 47]. While these approaches allow modeling of different NICs in a fixed system architecture, they do not lend themselves to modeling the range of system architectures explored in the coming chapters. Simulation provides this flexibility, including not only, the ability to investigate changes to the network interface as a simple peripheral device, but also the ability to reconfigure systems to allow the NIC to be a central system component. While many architectural simulators exist, the majority focus on the processor and/or memory system and do not provide operating system or I/O modeling support. Investigation of networking issues requires both the ability to incorporate the execution of operating system code and detailed performance modeling of the I/O subsystem. This set of requirements led to the development of M5.

Simulation, of course, is not without its drawbacks. The M5 simulator, like most simulators, incurs a performance penalty relative to other techniques such as emulation. This penalty is balanced by the reduced cost in terms of time and money in development and production of physical prototypes. To cope with the performance penalty, I use standard methods such as checkpointing and warm-up (further discussion in Section 4.4).

The majority of this chapter introduces the M5 simulator, investigates the need for full-system simulation, and describes the various components of the simulator having direct relevance to this thesis. Additionally, the chapter contains a synopsis of effort in validating M5's performance model and a discussion of related work in simulator systems.

3.1 Full-System Simulation

The traditional target of architecture research has been improving the performance of general-purpose user or scientific applications via architectural innovation. These applications spend a majority of their time in unprivileged code, and thus do not require extensive simulation of the operating system as it relates to hardware. Thus, most simulators today simulate in a stand-alone fashion, where the user-level code of a program is executed instruction-by-instruction in a detailed CPU simulator. Interactions with kernel software via system calls are functionally emulated and not actually executed on the simulator platform. In theory, it would be possible to extend this approach to do networking experiments, where socket `read()` and `write()` system calls are emulated to send data to the network device. However, a large fraction of network activity occurs in the kernel; thus using emulation would not provide even close to the level of detail necessary to achieve reliable results. Detail in simulation is necessary in the area being studied. Thus, network-centric system research necessitates a simulator capable of detailed full-system and networking subsystem simulation.

While a few full-system simulators exist [73, 53, 76], none provide the detailed network I/O modeling I require, and adding this functionality to those simulators would not have been straightforward. To address this, I decided to extend an existing application-only architectural simulator that my research group was using to meet my full-system needs. The resulting simulator grew in to what is now known as the M5 simulator system, a freely available simulator [52] with capabilities beyond what are used for this dissertation. M5 has ancestral roots in the SimpleScalar [14] simulator and though SimpleScalar code is no longer required, it does, as a result, execute 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.

SimOS/Alpha [73] was an invaluable aid in bringing up initial full-system support on M5. I used SimOS/Alpha as a reference platform for development, initially modeling the same functional pieces such as the TurboLaser platform and the Alpha 21164 processor, and supporting the same software, Tru64 UNIX v4.0. Although the OS and PAL code use 21164-specific processor control registers, the performance model more closely matches a 21264, including support for all user-visible 21264-specific Alpha ISA extensions. While SimOS/Alpha was able to boot the Tru64 Unix v4.0 kernel, it had become antiquated, so I then updated support to Compaq Tru64 UNIX v5.1. It then became apparent that Tru64 itself was limiting, so the decision was made to support Linux and FreeBSD.[1] Linux was desirable because it better facilitated kernel modifications as necessary for my research. Unfortunately, Linux does not support the TurboLaser platform, resulting in a switch to the Tsunami chipset. The switch to Tsunami enabled M5 to boot unmodified Linux, including kernels in the 2.4.x series and the 2.6.x series, as well as FreeBSD. For the results in this dissertation, I used Linux 2.6.13.

In addition to the above modifications, the detailed CPU timing model captures the primary timing impact of system-level interactions. For example, M5 executes the actual PAL code flow for handling TLB misses. For a memory barrier instruction, the pipeline is flushed and stalled 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, M5 models multiple systems and the network interconnecting them in a single process. While it isn't strictly necessary for these to all be in a single process, doing so facilitates them all sharing a single global concept of time. Having one notion of time is necessary for producing repeatable results. M5 has a notion of a fixed unit of time called a "tick". What unit of time each tick represents is configurable, but in general, having 1 tick represent 1 ps is sufficient for all our simulation needs. To facilitate the use of these ticks, M5's configuration system requires that all frequency and latency parameters be provided in terms of some real unit of time, such as μs, ns, or GHz. These parameters are then converted into a corresponding number of ticks for the desired latency or (for frequency parameters) clock period. The configuration system ensures that the tick can provide a resolution within a configurable error (0.1% tolerance is the default and was used for this dissertation). This global notion of time makes it very simple for M5 to support CPUs of different speeds that are not multiples of each other, as

[1]Researchers at the University of New South Wales in Sydney, Australia have additionally added support for the L4Ka::Pistachio μkernel.

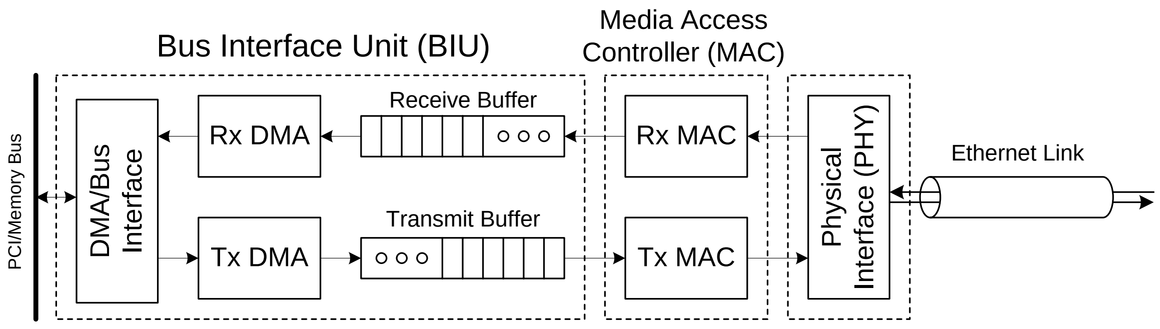


Figure 3.1: NIC Block Diagram - Block diagram depicting the high level components of a standard Network Interface Controller.

well as bandwidths and latencies that are not multiples of the CPU speed. This is something not typical of most architecture simulators whose notion of time is derived from the processor clock.

Implementing the capability to support multiple systems was simplified by the object-oriented design of the simulator—creating another system is simply a matter of instantiating another set of objects modeling another CPU, memory, disk, etc. All of these objects are also tied together by a sophisticated, object-oriented configuration language that allows you to group objects into larger units (e.g. entire systems), and instantiate multiple copies of those units. This high level of object orientation and organization is similar in spirit to Asim [27].

While there are a few other simulators that can do full-system simulation, M5 stands out with its detailed network device model, as well as the supporting device driver necessary to adequately perform experiments involving network-centric system design. Without this realistic device model, it would not be possible to have confidence in the results presented in this dissertation. An accurate device model is, however, not enough to get the whole picture. It is also important to pay particular attention to fully modeling the impact of the network device on the memory system and CPU via direct memory accesses (DMA) and programmed I/O (PIO). The final necessary piece required to accurately model a network device is an interrupt mechanism, such that when the modeled device receives a data packet, it is able to interrupt the processor much like a real-world system.

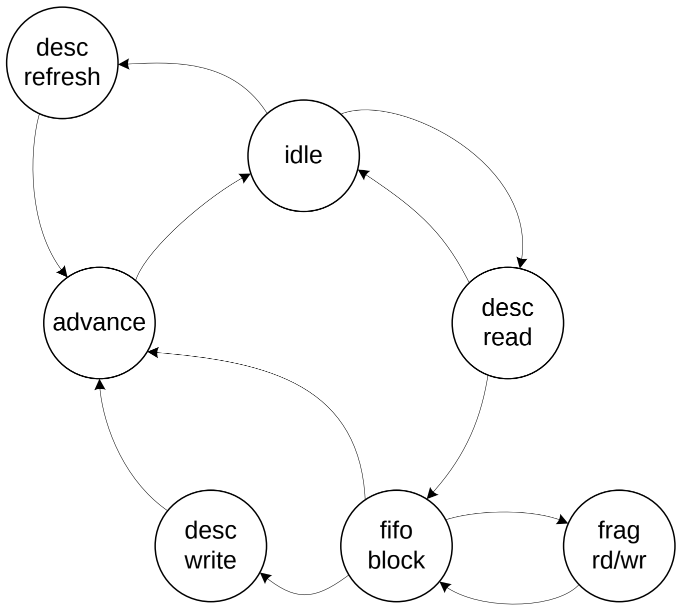


Figure 3.2: NIC State Machine - Simple state machine implemented by the National Semiconductor DP83820 Gigabit Ethernet controller. While this state machine is specific to that part, it is generally applicable to most Ethernet controllers available today.

3.2 Ethernet Device Models

Since the goal of this dissertation is to investigate the impact of new system architectures for networked systems, it was necessary to develop sufficiently detailed network device models. The block diagram of the design of the overall model is given in Figure 3.1. The Ethernet device models focuses its detail on the main packet data path and the I/O interface for programming the device. Four logical blocks of the model comprise the Ethernet model: the buffer interface unit, the media access controller (MAC), the physical interface (PHY), and the link.

There are three different NIC models provided with M5. The first one developed did not model any existing device; rather, it was a model of something that represented the general characteristics of commercially available NICs. It had a similar state machine (see Figure 3.2) to the National Semiconductor part described below, but it was only an approximate model. The motivation for developing this model was that the Ethernet NICs supported by Tru64 Unix, the original OS brought up on M5, did not have any public documentation. Since I designed a model for something that did not actually exist, it was necessary to write a device driver for the model in order to use the device. One benefit of this process is that because the device driver and device model were written at the same time, it allowed me to build an efficient interface between the two. When Linux was brought up on M5, rather than porting the original Ethernet device driver to Linux, the decision was made to model a real off-the-shelf device in M5. This decision led to the development of a second NIC model, which emulates the National Semiconductor DP83820 [61] Gigabit Ethernet device. The model is sufficiently faithful to support the off-the-shelf Linux ns83820.c device driver. I fixed a bug in the real DP83820 (and removed the corresponding workaround in the driver) that prevents its DMA engine from writing to arbitrarily aligned blocks.[2] The final model available in M5 is the SINIC/V-SINIC model used in the study in Chapter 6 and Chapter 7.

The buffer interface unit manages device programming registers, DMA to and from the device buffers, interrupts, and the assembling and buffering of packet data. I functionally model the PCI configuration register space to get device configuration information, but there currently is no explicit PCI model for timing. The device does, however, participate fully in the system bus protocol for both DMA transactions and programmed I/O requests to device control registers. As a result, memory transactions occupy bus cycles and cause coherence/invalidation traffic on the bus.

The MAC portion of the model simply moves data from the transmit buffer to the link via the PHY, or passes the data from the link via the PHY to the receive buffer. The MAC portion of the device model serves solely as an arbiter for the link and incurs no extra timing overhead. Additionally, the PHY portion of a real Ethernet device is responsible for converting between the electrical signaling domain of the device and the signaling of the actual link. Thus, the model does not have any timing information for the PHY.

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 I am essentially modeling a simple wire, only one packet is allowed to be transmitted in each direction at any given time. It is up to the sender to wait for the link to become free and up to the receiver to be ready when the packet arrives. If the sender does not wait, a collision occurs. If the receiver is not ready, the packet is simply dropped. In addition to the wire time, an additional delay parameter can be configured on the Ethernet link to represent the transit time across

[2]This common feature found in most Gigabit Ethernet NICs allows the kernel to align buffers based on packet payloads rather than headers. More discussion of this alignment is found in Section 2.3.2.

the Internet. This delay parameter does allow for buffering of packets representing the actual packets that would be in flight across a large network like the Internet.

3.3 Interrupts

The interrupt system modeled is relatively simple. In terms of timing, the interrupt signal propagates instantaneously. Functionally, there are a number of interrupt lines, each of which can be shared. Those lines go to an off chip interrupt controller which shares those lines with each device. When a device signals an interrupt, it propagates to the off chip controller which sends it to the CPU. At this point, the CPU jumps to interrupt handling code. The interrupt handler will then poll the controller to determine which device triggered the interrupt. The the handler will poll the device to find out what event occurred.

Older 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. There are several schemes that reduce the overhead of interrupts (see Section 8.4). I use a fixed-delay interrupt moderation scheme to reduce the actual number of interrupts posted. This scheme uses a timer to defer delivering a packet interrupt for a specified time (I use 10μs in the experiments presented in this dissertation). 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 Memory and I/O System Model

The memory and I/O systems are key determinants of networking performance. M5 provides a handful of simple memory system components—a cache, a bus, a bridge, and a memory—that can be composed to produce the desired system configuration.

M5 provides 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. The DRAM, NIC, and disk controller models incorporate a single slave interface to this bus model, capable of variable transaction sizes up to 64 bytes. 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.

M5's bus bridge model 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 and for the memory controller. In the 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.

To accurately model the impact of a device on the system as a whole, I model the PIO and DMA transactions that take place between the CPU, memory system, and the device. Like most real-world devices, the configuration of the device is accomplished by attaching the device to the memory bus, memory-mapping the device registers into a chunk of memory, and doing loads and stores from the CPU. Because these accesses are done with normal loads and stores from the CPU, they are issued by some program (usually the operating system) and as a result are called programmed I/O. For example, when a packet is transmitted, it is necessary to notify the device that there is data ready for transmission. This notification is accomplished simply by setting a control register bit on the device using a store instruction. Additionally, both PIO reads and writes can have side-effects and those side-effects are expected to happen when the instruction is executed. As a result, these memory locations cannot be cached by the memory hierarchy and every one must traverse the memory hierarchy to the device every time. Finally, because PIO reads can have side-effects, they are not considered idempotent, and can only be issued once the instruction is known to be non-speculative.

The alternative to PIO is direct memory access (DMA). DMA allows the device itself to directly read data from memory and write data to it. These reads and writes are set up by using programmed I/O to point the device at a memory location to access. The device uses the address that was written to the device via PIO to point to a well known data structure. That well known data structure is generally a linked list or array of descriptors (shown in Figure 2.3). The descriptors contain physical address, length, and flags fields. The address and length fields describe a chunk of memory to read from or write to, and the flags typically depict things such as the ownership of a descriptor (host vs. device) and error conditions.

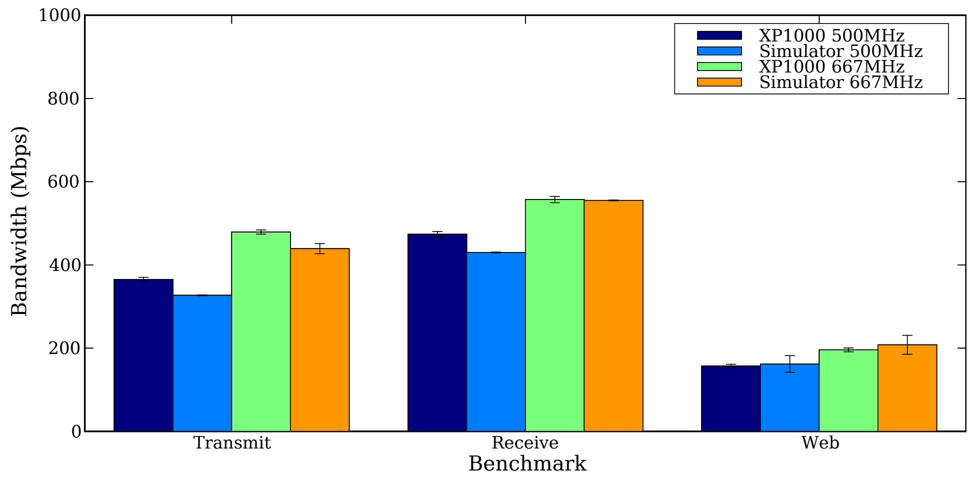


Figure 3.3: Achieved Bandwidth of M5 vs. Real Systems - M5 was tuned to have similar characteristics of two real-world systems. The achieved bandwidth of these systems is compared to the achieved bandwidth of M5 tuned to these systems.

3.5 Validation

While it was relatively easy to validate M5's functional model since incorrect functionality generally leads to incorrect program output or a program that just fails to even execute, performance validation is another thing all together. Without comparison to some standard, there is really very little that can be done to assure the user that the performance numbers reported by a simulator are actually correct. To this end, M5 was validated against two Compaq Alpha XP1000s (w/500MHz and 667MHz CPUs). The validation results, depicted in Figure 3.3 show that five of the six benchmark/system combinations tested have simulated network bandwidth within 15% of the real system. CPU utilization breakdowns (user, driver, stack, etc.) also correlate well. This level of accuracy is reasonable given that M5 does not try to model the XP1000 platform precisely. Rather, the generic CPU, cache, bus, and memory parameters were merely tuned to match. More information on the validation of M5 is available [74].

3.6 Related simulator work

There are a large number of simulators designed for architectural research, but only a limited fraction of them are capable of running operating system code, and none of them would allow me to easily accomplish this architecturally related networking research I am interested in doing. The reason for the lack of both of these features is simply a matter of difficulty and time investment. First of all, full-system simulation, required to execute operating system code, requires at least as much, if not more, of a time investment as implementing the user visible instruction set and the CPU and memory system timing models. Additionally, to implement functionality to examine network workloads, it is necessary to be able to simulate two complete systems in one timing domain in order to achieve deterministic results. This in itself is not a huge task, but it does require that the overall design of the simulator be flexible enough to allow such a thing. Finally, an additional component required for network workloads is that the I/O subsystem have a proper timing model. This is also not a particularly difficult add-on, but it requires that the memory system is flexible enough to allow such a thing.

SimOS [73] was the first simulator to enable operating system code to be simulated for architectural research. In addition, SimOS provided the user the ability to trade off simulation speed for accuracy between simulations, and even within a simulation by providing on-the-fly switching between models with different levels of detail. The reason for not starting with SimOS with this research was that I already had a detailed CPU simulator with SMT support and a detailed memory model. The choice was to either graft the models into SimOS, or to add the OS capability to our existing simulator. I determined that the latter option would be simpler. It should be said though that SimOS was invaluable while adding OS support because it was possible to use it as a "golden brick" for instruction-by-instruction comparison. Other related simulators include: SimICS [53], a functional simulator whose commercial nature prevented me from being able to achieve me goals; TF-Sim [54], which is a timing simulator grafted onto SimICS; ML-RSim [76], another full-system simulator designed for studying I/O which was not available when I started M5[3]; Asim [27], an internal simulator at Intel which is not publicly available; and SimpleScalar [14] which, while

[3]While ML-RSim now has much of the functionality of M5, it is lacking in several areas in which M5 excels: (1) it is unable to simulate multiple systems within one timing domain; (2) it cannot run unmodified operating systems and can only execute Lamix, a modified version on NetBSD; and (3) its memory hierarchy is fixed.

not providing operating system support, did provide a starting point for the detailed CPU model.

3.7 Summary

To enable this research, I extended the M5 simulator to be capable of simulating a complete computer system, including not only CPUs, caches, and memory, but also disks, network interfaces, and other peripherals. The simulator simulates hardware with enough fidelity to boot several unmodified operating systems, including: Linux, FreeBSD, the L4 μkernel, and Tru64. M5 is also capable of simulating multiprocessor systems. There are a few key capabilities of the M5 simulator that make it unique. One is that it can simulate multiple systems in a single process. In other words, with one instance of the running binary, I can simulate two complete networked systems and take statistics on the nature of their interactions. This enables each simulation to be deterministic and repeatable—an essential feature enabling the user to attribute changes in data to changes in architecture and not spurious simulation differences. Since detailed full-system simulation understandably takes longer than non-full-system simulators, M5 supports checkpointing in order to be able to jump start simulations at interesting portions of code. For example, booting the operating system has no bearing on the performance of the networking applications I generally run atop the operating system, and thus should be ignored when analyzing data. Like most simulators, M5 has two models of CPU, a simple fast one and a detailed slower one. M5 can be configured to run the simple CPU and dump all the state that is relevant to a full-system run at arbitrary pre-specified cycles. Once this has happened, M5 can run simulations beginning from any of these interesting points, and thus skip unimportant portions such as boot. Of course, the simple CPU does not provide much insight into benchmark performance, so M5 also has the capability to switching from simple to detailed CPU's in the middle of a simulation. After beginning from a checkpoint in simple CPU format, it can run for a period of time in simple CPU to warm up the memory system, and then switch to the detailed CPU to obtain the truly relevant data I desire without having to wait for the beginning of the simulation to execute. This provides a powerful mechanism to cope with the performance overhead of detailed system simulation.

Chapter 4

Analyzing Network Workloads

This chapter begins with an examination of the characteristics of TCP that must be considered when studying network workloads under simulation. This explanation is followed by a description of the benchmarks used for the studies in this dissertation. Next is some detail on how the M5 simulator was configured to run the simulations, including a discussion of how the memory latency numbers were chosen. Finally, this chapter concludes with a discussion of the methodology employed in all of the studies.

4.1 TCP Stability

Because all of the benchmarks described below use TCP, an examination of the simulation characteristics of TCP is necessary. Two factors come into play when simulating TCP. First, TCP is specifically designed to tune itself to the performance characteristics of the underlying system and network. This tuning is built into the protocol as part of its congestion control and flow control mechanisms. Second, due to the major speed penalty of simulators, simulator users have developed several techniques to cope with the slow execution time of benchmarks. These techniques include: reducing benchmark size, executing code in fast-forwarding mode to get to interesting regions of a benchmark, and sampling.

Because of the self-tuning nature of TCP, and the inherent performance instability of the time-saving techniques described above, care must be taken to ensure that simulations are properly conducted and result in useful executions. Each one of the techniques described above provides its own challenge to network simulation. Reducing the benchmark size is difficult for simulation. Even a small benchmark requires the operating system, an enormous program by any standard. This fortunately does not present TCP itself with a major problem. Fast-forwarding, on the other hand, presents challenges due to its interaction with the networking protocols being tested. Because TCP is specifically designed to tune itself to the performance characteristics of the underlying system and network, when switching from a fast-forwarding mode to a detailed simulation mode, those characteristics change, causing TCP to adapt. One side effect of this change, typically seen when the fast-forward portion has a higher performance than the detailed portion, is that the second execution portion will be overwhelmed by the packets sent before the transition and packets will be dropped as a result. Due to TCP's slow recovery from dropped packets and the very little amount of real time that can be simulated, a single dropped packet can ruin an entire simulation. This high to low transition seen above makes normal sampling techniques difficult because the simulation cannot switch between different CPU models without perturbing the performance and risking packet loss. Instead, the simulation must either be started from the beginning and fast-forwarded to each sampling point, or checkpoints must be taken at each sampling point. Either way, the normal cache warm-up phase must be re-executed. A detailed discussion of these effects can be found in another publication [37].

During the warm-up phase, the system under test was simulated with a simple one cycle-per-instruction CPU with a blocking cache, whereas, the sampling phase uses a four wide detailed out-of-order CPU with a non-blocking cache. This ensures that the TCP protocol behaves in a representative fashion because the warm-up periods of the experiments are of lower effective performance than the detailed simulations following, allowing the simulations to adapt quickly to the bandwidth change.

4.2 Benchmarks

All the benchmarks—Netperf, SPEC WEB99, iSCSI, and NAT—used for evaluation in this dissertation were simulated in a client-server configuration where the system under test, either client or server depending on the benchmark, was modeled in detail. The stressor, on the other hand, was modeled functionally with a perfect memory system so as to not be the bottleneck.

4.2.1 Netperf

Netperf [36] is a network micro-benchmark suite developed at Hewlett-Packard. It contains a variety of micro-benchmarks for testing the bandwidth characteristics of sockets on top of TCP or UDP. Out of the available tests, I use the TCP stream benchmark, a transmit benchmark, and the TCP maerts benchmark, a receive benchmark. In both these cases after setting up a socket one machine generates data as fast as possible by calling `send()`. Normally this call returns as soon as the data is copied out of the user buffer. However, if the socket buffer is full, the call will block until space is available. In this way the benchmark is self-tuning. Similarly the second machine attempts to sink data as fast as possible by calling `recv()` continuously. In general, the time spent executing the benchmark code is minimal, and most of the CPU time is spent in the kernel driver managing the NIC or processing the packet in the TCP/IP stack.

In addition to the standard stream and maerts benchmarks, my research group developed a variant of each that uses more than one TCP connection, with the sender and receiver spreading packets across the connections in a round-robin fashion similar to Hacker et al. [33]. These benchmarks, which I term stream-multi and maerts-multi, provide more streams for the kernel and NIC to manage while guaranteeing that they are all of similar bandwidth. With several independent streams this isn't the case, because one stream could end up getting the majority of the available bandwidth it would be harder to have compare different executions of the micro-benchmark due to different numbers of active connections.

4.2.2 SPEC WEB99

SPEC WEB99 [81] is a popular benchmark that is used for evaluating the performance of webservers. The benchmark simulates multiple users accessing a combination of static and dynamic content using HTTP 1.1 connections. In my simulations I used Apache 2.0.52 [3] with the mod_specweb99 CGI scripts. These scripts replaced the reference implementation with a more optimized implementation written in C and are frequently used in the results posted on the SPEC website.

The standard SPEC WEB99 client isn't well suited for a simulation environment. A standard SPEC WEB99 score is based on the maximum number of simultaneous clients that the server can support while meeting some minimum bandwidth and response time guarantees. Each client requests a small amount of data and thus a SPEC WEB99 score is normally obtained by a large testbed of clients and an interactive tuning process so as to find the maximum number of clients for a particular server configuration. This approach isn't practical for my tests because of the large slowdown simulation incurs. For my purposes I am not concerned with the SPEC WEB99 score attainable by the machine under test, but rather are simply interested in the performance characteristics of a web server workload. To this end I chose to use a different client based on the Surge traffic generator [5] that preserves the same statistical distribution as the original client but is able to scale its performance up to a point that it can saturate the server.

4.2.3 iSCSI

iSCSI [75] is a relatively new standard for network attached storage. The data movement portion of the protocol is simply SCSI on top of TCP/IP. An initiator (client) is able to access a target (server) in a similar manner as it would locally with a SCSI host adapter connected to a SCSI device. Because of its use of TCP/IP as a connection layer protocol and Ethernet as a link layer protocol commodity networking hardware can be used, making it much cheaper than traditional network attached storage systems (e.g. FibreChannel).

All of the iSCSI tests in this dissertation used the Open-iSCSI initiator and the Linux iSCSI Enterprise target. The target was configured to have a null backing store, making it return random data immediately, a reasonable simplification since I am not concerned with disk I/O performance. On top of the iSCSI client I run a custom benchmark that uses Linux's asynchronous I/O (AIO) interface to sustain multiple outstanding reads to the iSCSI disk. In this benchmark, as soon as read completes, a read to a new location is initiated. It is possible to study both the performance of the initiator and the target in this configuration, but for the experiments done in this dissertation, only the initiator is examined. As stated above, the system under test is simulated in a detailed fashion with the driving system configured in such a way as to never be the bottleneck.

4.2.4 Network Address Translation Gateway

The final benchmark used in this dissertation is that of a network address translation (NAT) gateway. NAT [26] is the process of translating network addresses from one namespace to another. NAT is typically used in the case where the gateway has a public IP address and is multiplexing many hosts hosts with private IP addresses onto that public address. This can be done for security since the private addresses cannot be addressed by machines outside of the private network, and are as a result more difficult to attack. It can also be done as a way to cope with the shortage of public IP addresses on the Internet. The NAT benchmark has a slightly different configuration than those above. First, three systems are necessary: the client, the

Table 4.1: Simulated System Parameters - CPU and memory system parameters used for all studies in this dissertation.

Frequency	2 GHz, 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, 16 MSHRs Inst: 1 cycle hit latency Data: 3 cycle hit latency
L2 Unified Cache	varies, 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
CPU Die Bus Bridge	75ns
Chipset Bus Bridge	90ns
I/O Bus Bridge	180ns
Main Memory	65 ns latency for off-chip controller, 50 ns on-chip

server, and the gateway. The client and server for this setup are running the maerts test described above. Both client and server are simulated in the artificially fast mode to ensure that they are not the bottleneck. The gateway is configured just at the test systems are in the other benchmarks, except that it has two NICs, representing the two interfaces commonly found on this sort of machine.

4.3 Simulator Configuration

All of the experiments in this dissertation used the M5 simulator, as described in Chapter 3. In this section, I will describe the specifics of how M5 was configured to conduct those experiments.

Table 4.1 lists the system parameters used for my simulations. These general parameters are intended to represent the characteristics of a modern high performance processor. Because the Linux protocol stack is not sufficiently parallel to enable it to take full advantage of future chip multiprocessor systems, the variations in frequency represent the increased performance that should be had through future chip multiprocessor systems and improved parallelism in the Linux kernel.

Table 4.2: Warm-up & Sampling Intervals - The length of time in processor cycles spent in the warm-up and sampling phases of simulation. The warm-up phase used a functional CPU in order to reduce simulation time, while the sampling phase used the detailed CPU model for proper timing. All results in this dissertation were taken over the sampling interval.

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

Because I desire to model high performance processors that are available today and in the near future, I configured the latency of memory, bus bridges, and peripheral devices to match numbers measured on modern servers. In order to do this, it was necessary to measure the memory latencies of various memory accessible devices. Some of the final parameters shown above are approximations, but because I am primarily interested in the relative behavior of these systems rather than their absolute performance, this is acceptable. Among the parameters shown above, are three bus-bridge latencies which were used for the various bridges found in the different configurations shown in Section 5.1.

Lastly, for the experiments in this paper I used a 1500 byte maximum transfer unit (MTU) as it is the standard on the Internet today. Although changing it is reasonable in a controlled environment, it won't be used for commodity traffic on the Internet and thus I fix it in my experiments.

4.4 Simulation Methodology

The time required for full-system cycle-level simulation is immense and, in general, is several orders of magnitude slower than a real system's performance. Thus running any benchmark to completion is impractical. To address this I use a combination of functional fast-forwarding, warm-up, and sampling to obtain the results herein. In order to fast-forward to an interesting point in a benchmark, I inserted a special M5 specific instruction in each benchmark that causes the simulator to checkpoint its state at a specific time. I use these instructions to take checkpoints only after the workload enters its steady state. From each checkpoint I warm up the state in the caches and TLB using a simple CPU model. After the warm-up is complete I switch to a detailed cycle-level CPU model and run the experiment. The amount of time spent in the warm-up and sample phases is benchmark dependent since each benchmark has different stability characteristics. Table 4.2 shows the parameters used for each benchmark.

The warm-up and sample phase times were chosen based on the stability of the overall bandwidth of each benchmark. To determine this stability, each benchmark was run for 10 billion cycles in the detailed simulation mode. During this long run, statistics were taken at 10 million cycle intervals. These small samples were aggregated into larger intervals until the variation between intervals was within the desired limits of 5% for the macro-benchmarks and under 1% for the micro-benchmarks. The sampling time values shown in Table 4.2 represent the interval size required to achieve these limits. The warm-up period was selected to ensure that the shorter run achieves the expected result.

Chapter 5

Integrating the Network Interface Controller

In this chapter, I consider the integration of the NIC onto the CPU die as depicted in Figure 5.1. Although integration is not common on high-performance servers today, there are numerous examples in the embedded space (e.g., from Broadcom [12]) where it is done as a cost saving measure. In the supercomputing realm, the BlueGene/L incorporates an integrated 1 Gbps Ethernet NIC for I/O [64] into each node. Given available transistor budgets, the potential performance benefits that will be described in this chapter, and the importance and ubiquity of high-bandwidth Ethernet, NIC integration is an obvious evolutionary step in the Internet server domain as well. Future revisions of Sun's Niagara product line are rumored to include one or more integrated 10GigE NICs [22].

This chapter covers the first major study of this dissertation where I use several network-intensive benchmarks (described in Chapter 4) to investigate the impact of NIC/CPU communication overheads. In this study, I analyze the performance of hypothetical systems in which the NIC is more closely coupled to the CPU, including integration on the CPU die. The study uses the benchmarks to investigate the impact of varying levels of NIC integration on achieved bandwidth, miss rates, and CPU utilization. Additionally, system parameters such as CPU frequency and cache size are varied to observe the effect on future system designs. I 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.

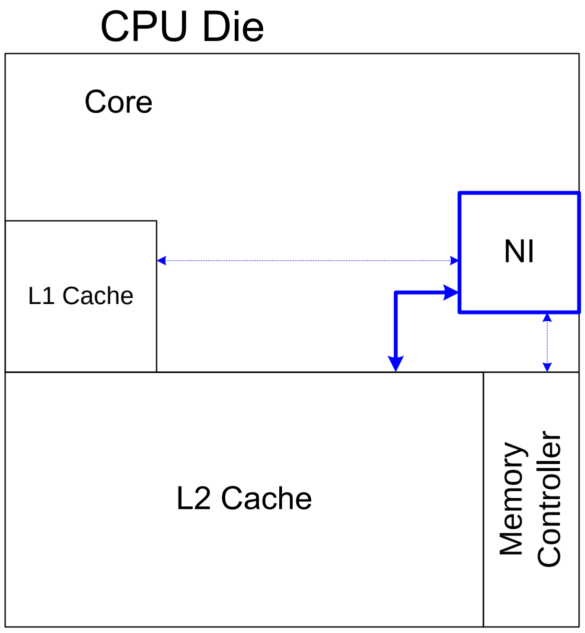


Figure 5.1: NIC Integration - With NIC integration, the entire NIC is simply put directly onto the CPU die. This placement allows the NIC to attach to the various memory structures found on the die.

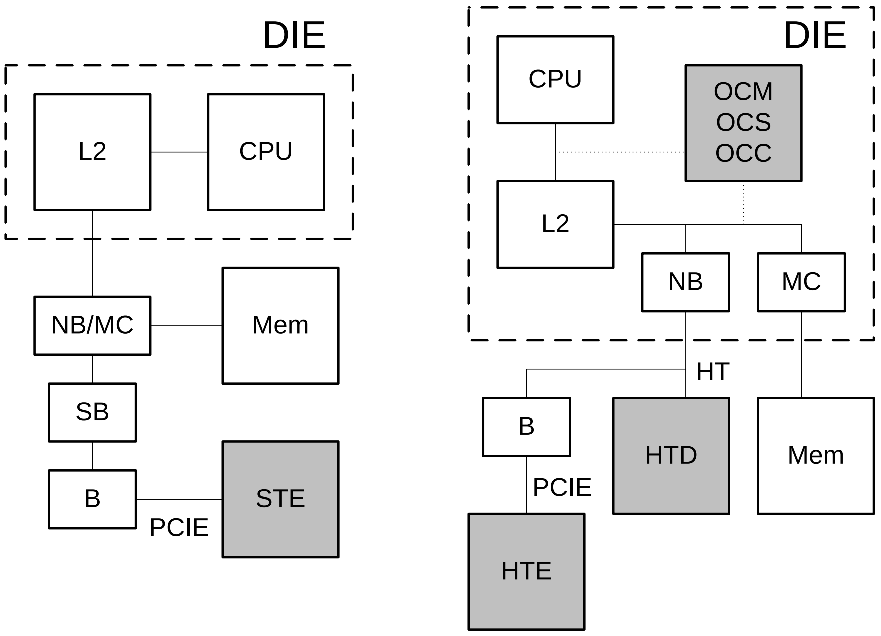


Figure 5.2: NIC Placement Options - Each gray box depicts the potential location of a network interface controller in the memory hierarchy. These locations are known as: (STE) Standard PCI Express, (HTE) HyperTransport PCI Express, (HTD) HyperTransport Direct, (OCM) On-chip/Memory Attached, (OCS) On-chip/Split Attachment, and (OCC) On-chip/Cache Attached.

5.1 NIC Placement Options

To investigate the impact of changing the location of the NIC in the memory hierarchy, I chose a set of five configurations as shown in Figure 5.2. The first two configurations model aggressive I/O designs that are expected to be common in the near future. 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 second system, the HyperTransport[1] PCI Express (HTE) configuration, represents a similar system with an on-chip memory controller, but with one fewer chip separating the NIC from the CPU.

The third configuration, HyperTransport direct (HTD), models a potential design for systems that implement a high-speed I/O interconnect via HyperTransport-like channels. This configuration is similar to attaching the NIC directly to a 6.4GB/s HyperTransport channel.

The remaining configurations integrate the NIC onto the CPU die. Three on-chip configurations are considered: on-chip memory-bus-attached (OCM), on-chip cache-attached (OCC), and on-chip split (OCS). OCM attaches the NIC to the memory bus, on the other side of the L2 cache from the CPU. This configuration provides very high bandwidth similar to HTD but even lower latency due to the elimination of a chip crossing. OCC goes one step further and attaches the NIC to the bus between the L1 and L2 caches. This configuration reduces latency even further, but more importantly allows incoming NIC DMA data to be written directly into the L2 cache. OCS is a hybrid of the two configurations, where the NIC is logically attached to both the cache bus and the memory bus. In this configuration, PIO accesses occur on the cache bus while DMA occurs on the memory bus. Finally, a configuration not depicted on the diagram, On-Chip/Split Payload (OCSP), is found in all of the experiments. OCSP has the same memory attachment as OCS, but instead of all DMA going over the memory bus, the NIC splits header from payload data (actually the first cache block from the remainder of the packet) and places the former in the cache and the latter in memory. The goal is to provide low-latency access to the header control information, which is likely to be processed quickly by the kernel, without polluting the cache with large data transfers that may not be referenced as quickly.

[1]True HyperTransport is not modeled here; the name is used simply convey the properties of the configuration.

5.2 Opportunity Cost

I believe the area and complexity required for integrating the NIC onto the CPU die are modest, but they do have a cost. Cost can be measured in many ways, but I will focus on die area and pin bandwidth. Couple the meager area requirements of a NIC with the overabundance of transistors available on a CPU die, and the area cost should be considered minimal. While I offer no hard numbers, it is certainly true that a conventional NIC has a significantly smaller area when compared to a general purpose CPU because of its drastically reduced complexity. The only required on-chip components are those which enqueue and dequeue packet data into and out of the network FIFOs and the I/O bus interface. The physical layer signaling logic, or even the bulk of the FIFO buffer space could remain off chip.

Perhaps more important is the pin bandwidth requirement for the NIC. In terms of bandwidth, one must first realize that the data must get on chip anyway. With an integrated memory controller, a DMA from a peripheral to memory requires twice the CPU bandwidth. Signaling between the on-chip and off-chip portions of the NIC could use dedicated pins or could be multiplexed over an I/O channel (e.g., HyperTransport). This setup 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. An integrated NIC can in fact cut the bandwidth in half if the data is brought into the cache and there is no cache miss. If pin count is an issue, that can be handled by multiplexing the data over an existing interconnect such as HyperTransport since the majority of interconnects are packet based anyway.

It is certainly true that integrating a NIC onto a CPU die makes that die more complex than if there were no NIC at all. I believe the benefits of integration outweigh the drawbacks.

5.3 Memory Latencies

Table 5.1 presents timings for memory locations on three real machines. These measured latencies were used to tune the bridge latencies found in Figure 5.2 as discussed in Section 4.3. The latencies were measured on real systems using hardware performance counters and a custom Linux kernel module. The module used the CPU

Table 5.1: Latencies - The latencies measured for memory locations at different positions of the memory hierarchy (all latencies in ns).

Location	Alpha DP264	Pentium III	Pentium 4	M5 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	

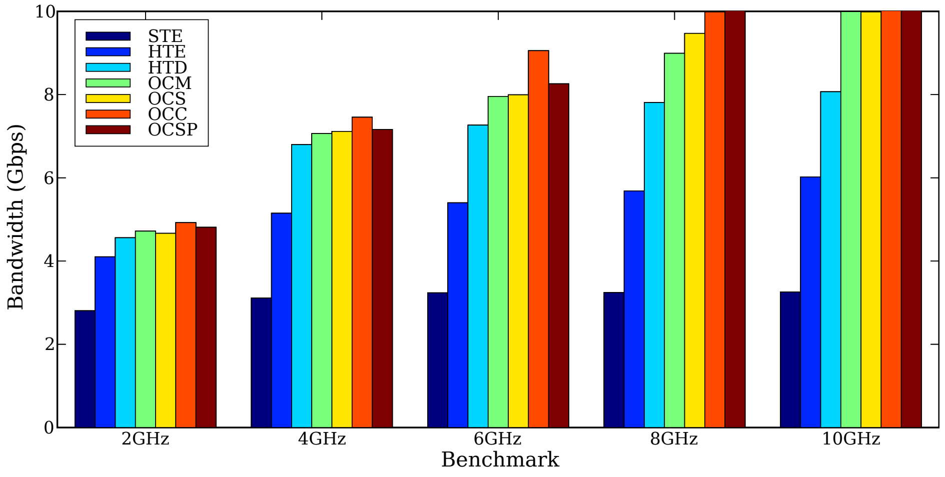
cycle counting instructions over many iterations of a loop accessing those memory locations to determine the average access latencies.[2]

Table 5.1 presents timings on three machines for accesses to devices in various locations throughout the memory hierarchy. The "peripheral" timing corresponds to the STE configuration, where the device is on a commodity I/O bus with multiple bus bridges between the device and the CPU. The "off NB" (north bridge) location is similar to the configuration, where a standard I/O bus is connected to the NB directly. The HTD configuration could be realized by integrating the NIC onto the NB ("on NB").[3] The timing for OCM is approximated by measuring the latency to a device integrated on the CPU die. The Alpha EV6 has no such devices, but both Pentium chips have an integrated local interrupt controller. These devices have an access time of 3x–4x more than the on-chip access time I modeled. However, I 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.

The memory access latencies for on-chip and off-chip memory controller were measured. With my 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 the results are similar to published numbers of 50.9ns and 72.6ns respectively [49].

[2]In conjunction with the cycle counting instructions, serialization instructions (x86) or false dependencies (Alpha) were inserted to insure that the cycle counter is not read until the access has completed.

[3]It is interesting to note that the peripheral and off-NB latencies in Table 5.1 have not varied by more than 10% over several years. Even the on-NB latency is only reduced by approximately 2x for more than a 3x change in CPU frequency.



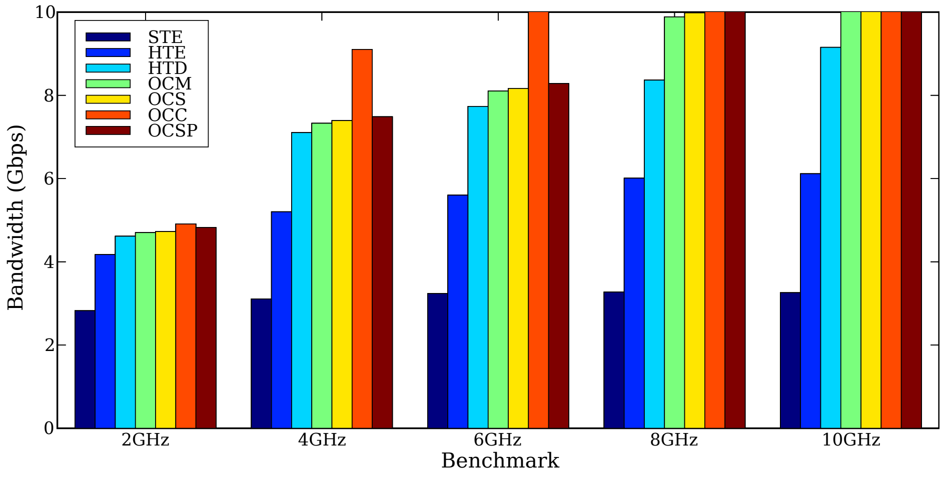


Figure 5.3: TCP Receive Micro-benchmark Achieved Bandwidth - The achieved bandwidth for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size (top), and a 16 MB level 2 cache size (bottom).

5.4 Micro-benchmark Results

I first examine the behavior of the simulated configurations using the Netperf micro-benchmark, varying CPU speed and L2 cache size. I then use the application benchmarks (web, iSCSI, and NAT) to explore the performance impact of the system configurations under more realistic workloads.

Figure 5.3 plots the achieved bandwidth of the TCP receive micro-benchmark across the 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. In the future, their

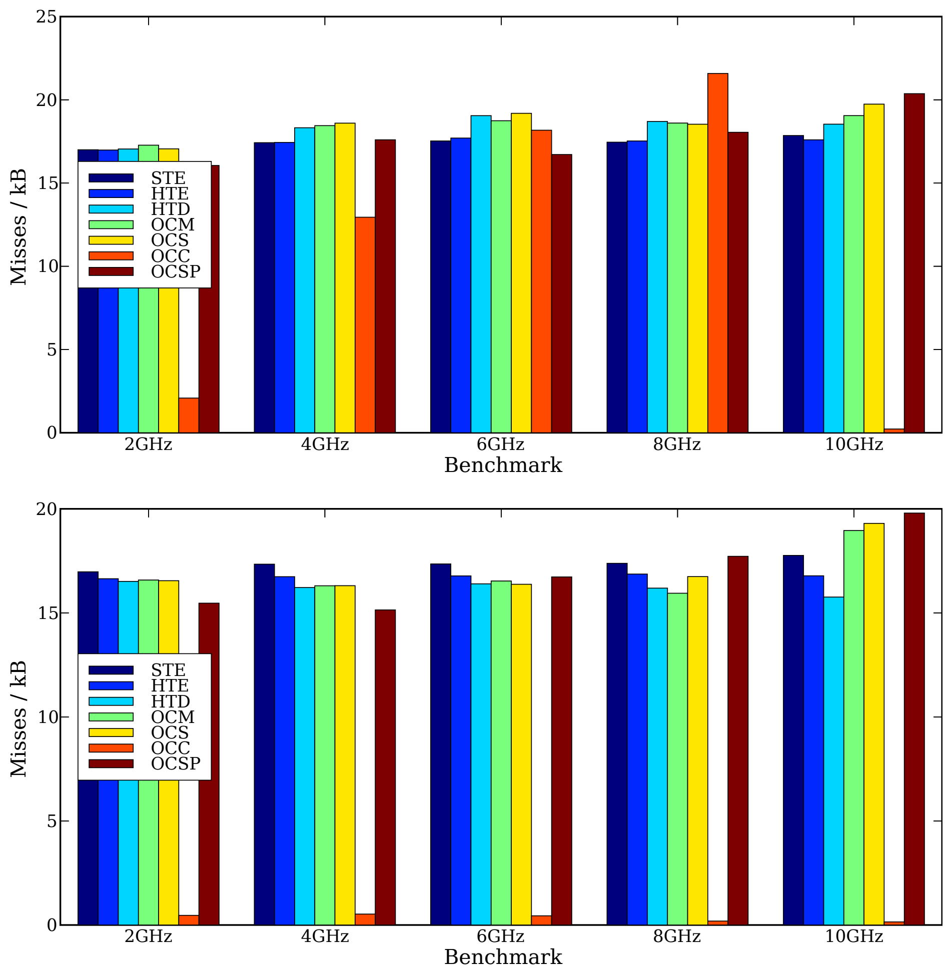
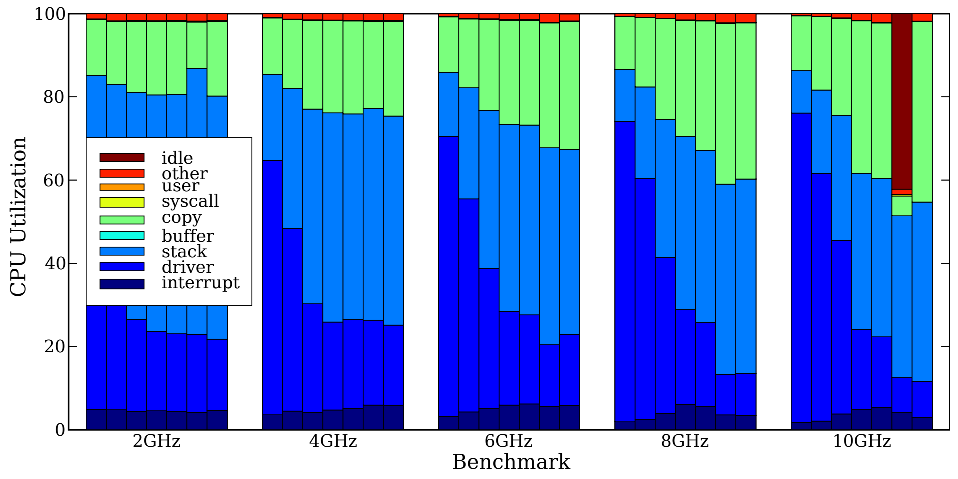


Figure 5.4: TCP Receive Micro-benchmark Cache Performance - Cache performance in number of misses per kilobyte of data transferred by the network interface for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size (top), and a 16 MB level 2 cache size (bottom).



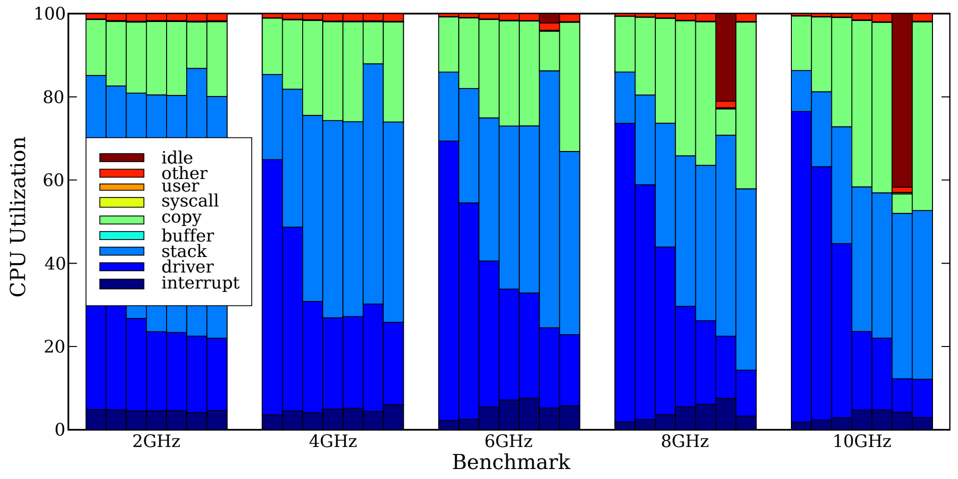


Figure 5.5: TCP Receive Micro-benchmark CPU Utilization - CPU utilization breakdown for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size (top), and a 16 MB level 2 cache size (bottom).

comparative performance will likely be achievable through multiprocessing, either through plentiful coarse-grained connection-level parallelism, as is seen in the macro-benchmarks below, or through advancements in protocol stacks that enable the processing of packets in parallel.

I first note that the STE configuration has no CPU bottleneck given that an increase in CPU performance does not impact bandwidth. It is likely that the latency between the CPU and the NIC alone accounts for the majority of the bottleneck given the fact that the HTE configuration also has nearly flat performance, albeit at a higher level. I draw this conclusion because the bandwidth of the PCI Express link in both the STE and HTE configurations is 1.067 GB/s (8.533Gbps) which is enough to allow for a higher network bandwidth. As opposed to the traditional configurations like STE and HTE, the integrated NICs universally provide higher performance at higher CPU speeds, though their advantage over the direct HyperTransport interface is slight at lower frequencies when the benchmark is primarily CPU-bound. Comparing the top and bottom graphs in Figure 5.3 shows that the more tightly coupled interfaces also get a larger boost from larger last level cache 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 5.4, which shows the number of last-level cache misses per kilobyte of network bandwidth for these configurations. Because OCM and OCS NICs write payload data to memory, the CPU will always miss in the cache when accessing that data. The result is a minimum of 16 cache misses per kilobyte of data transferred, as seen in the configurations without cache placement. Cache placement alone is not a guarantee that these misses can be alleviated since it is necessary for the cache to be large enough to hold the network receive buffers until the CPU accesses them. In this case, 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 requires less buffering. Because the micro-benchmark does not perform any application-level processing, the cache pollution induced by OCC when the cache is too small does not negatively impact performance.

Figure 5.5 presents the breakdown of 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, the most dominant bottleneck, as it reduces the latency of accessing device registers. This translates directly to the higher bandwidths of Figure 5.3, although most cases are still CPU-bound as the driver cost is replaced by CPU time in copy due to increased network bandwidth. On the other hand, when OCC is able to eliminate cache misses on network data (only with the 10 GHz CPU on this 4 MB cache), it drastically reduces the time spent in copy because the source data is in the cache rather than in memory. Looking at Figure 5.3 and Figure 5.5 together shows that although some other configurations manage to saturate the network, only OCC at 10 GHz does this with significant CPU capacity remaining to perform other tasks.

Figure 5.5 also illustrates 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. It is likely that this issue can be addressed using a more sophisticated interrupt moderation scheme. More information on interrupt moderation can be found in Section 3.3 and Section 8.4.

Figure 5.6 presents the performance, cache, and CPU utilization results for the TCP transmit micro-benchmark at a 4MB last-level cache size. In this micro-benchmark, the sending CPU does not touch the payload data. The result is similar to what one might see in a static-content web server. Since the payload is not touched, a larger cache does not affect the results.

At lower CPU frequencies, on-chip NICs exhibit a noticeable performance improvement over direct HyperTransport. Further, because transmit is interrupt intensive, low-latency access to the NIC control registers speeds processing. Again, one can 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 placement affects only headers and acknowledgment packets, giving OCC and OCSP similar behavior. Both incur significantly fewer misses relative to OCM, though this translates to only a slight decrease in CPU utilization due to the already low absolute miss rate; note the difference in scale on the misses/KB graphs of Figures 5.4 and 5.6.

5.5 Macro-benchmark Results

While the micro-benchmark results provide valuable insight into the fundamental behavior of the configurations, they do not directly indicate how these configurations will impact real-world performance. To explore that issue, I ran the three application-

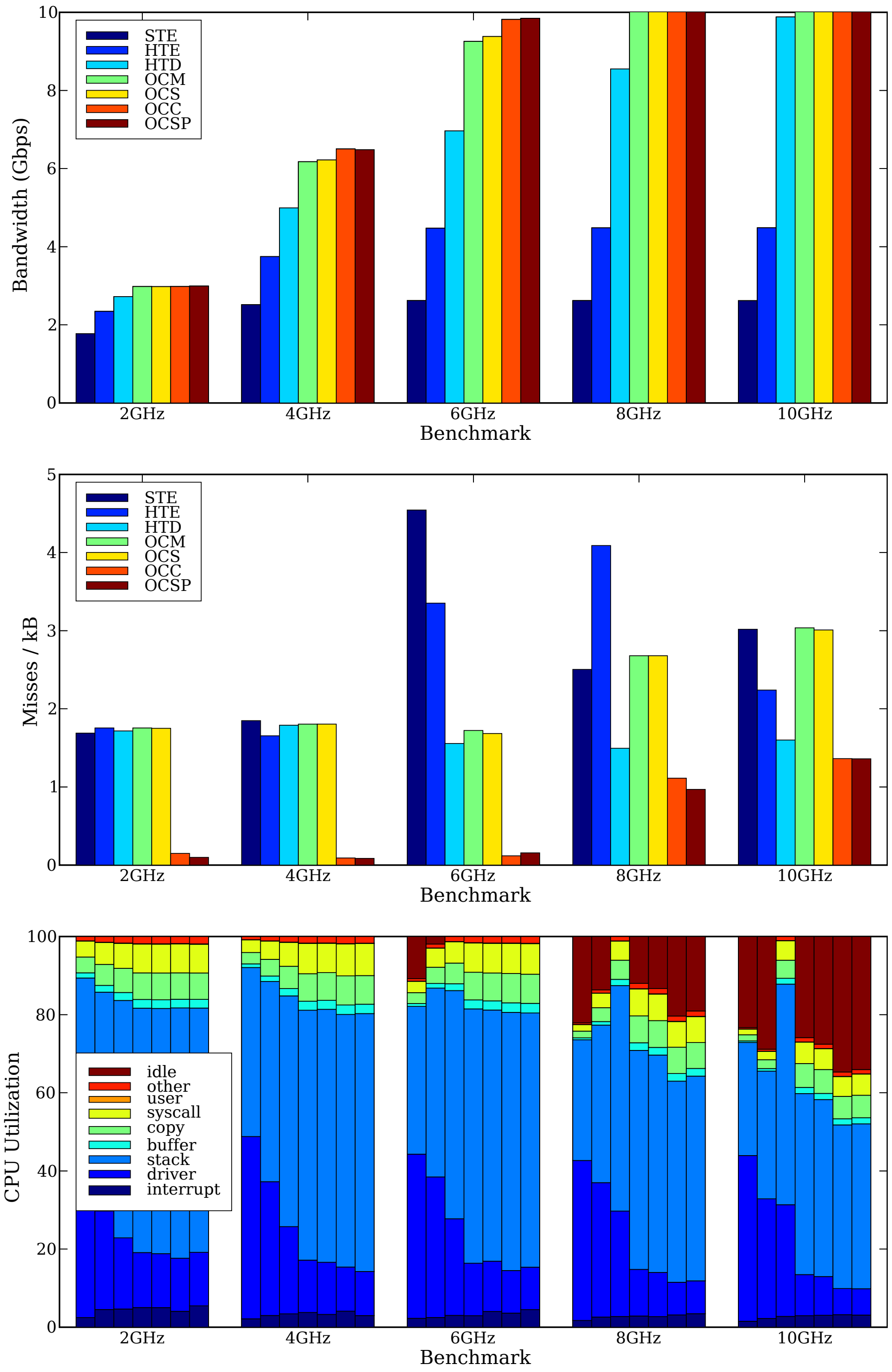


Figure 5.6: TCP Transmit Micro-benchmark - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size.

level benchmarks described in Chapter 4: the Apache web server, an iSCSI initiator, and a NAT gateway. Although I ran with both 4 MB and 16 MB caches, I present only the 4 MB results here. The 16 MB results can be found in Appendix refappendix:figures. For each benchmark, I 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 5.7. In this test 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 web server benchmark is highly parallel, and this single-CPU 10 GHz system performance could likely be achieved by a chip multiprocessor system. Since webserving is a largely transmit oriented benchmark, OCC's impact on cache misses is not very significant. The decrease in cache miss rate is largely due to a reduction in misses for ACK traffic and request traffic. Since both of these types of traffic tend to have very small packet sizes, we see a very similar miss rate for OCSP.

Figure 5.8 shows iSCSI initiator performance 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. Since the iSCSI workload in this case is read oriented, like the TCP receive workload and unlike the web server workload, iSCSI cache performance is greatly improved by the direct placement of data in the cache. However, unlike TCP receive, this does not translate into an actual bandwidth improvement because the cache miss rate is not high to begin with. 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. One significant issue with this benchmark is the high fraction of idle time. This is likely due to some improper configuration of the driver side of the benchmark, and translates closer integration into a larger fraction of idle time instead of an improvement of overall bandwidth. I believe that if the CPU utilization were maximized that larger performance gains would be seen for the integrated solutions.

Figure 5.9 shows the NAT gateway performance. This case shows the TCP receive micro-benchmark running between two hosts on either side of the gateway. The poor performance in the slower configurations is due to poor behavior under overload conditions. Because the NAT gateway system both transmits and receives data inten-

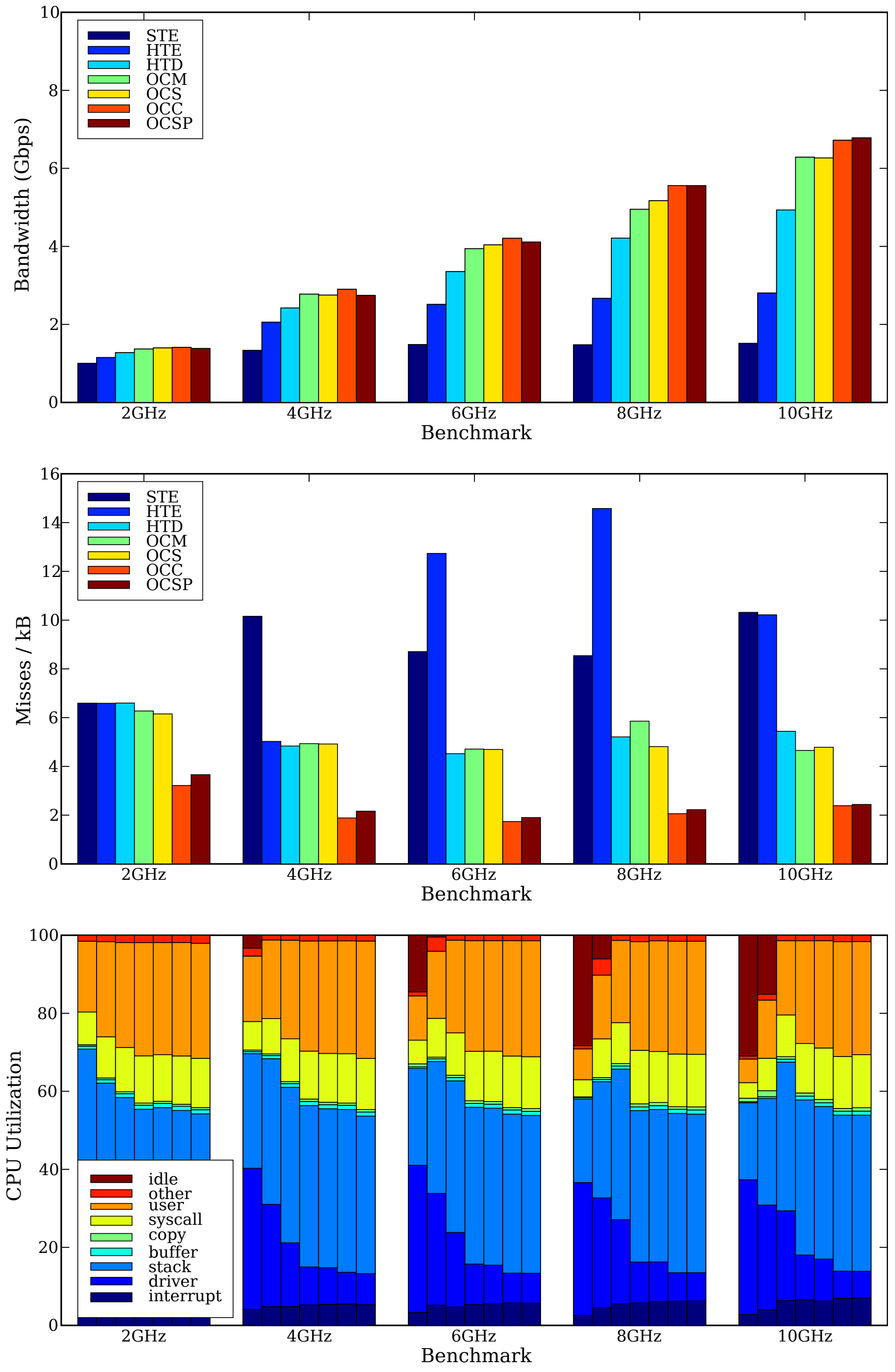


Figure 5.7: Web Server Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size.

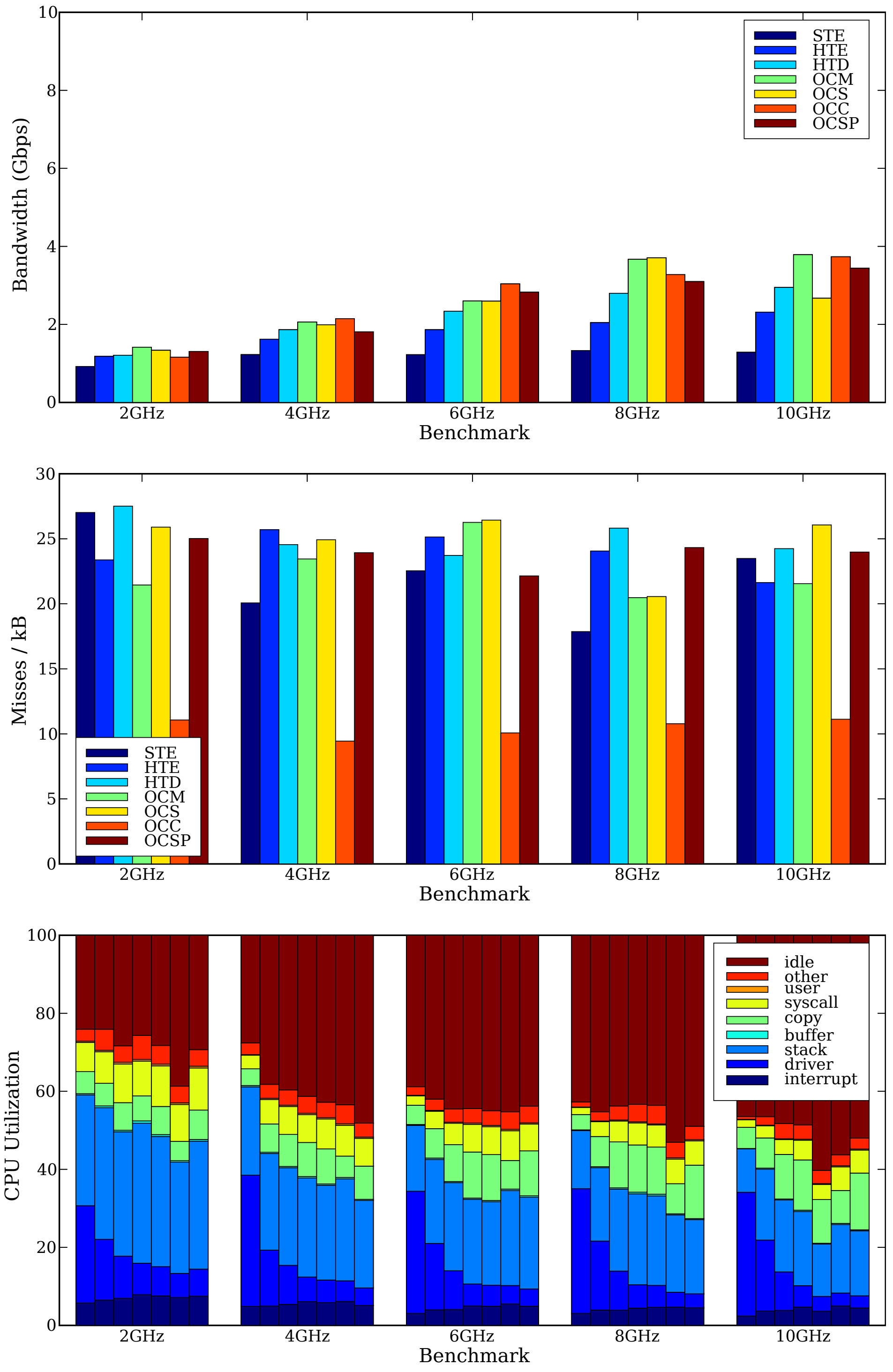


Figure 5.8: iSCSI Initiator Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size.

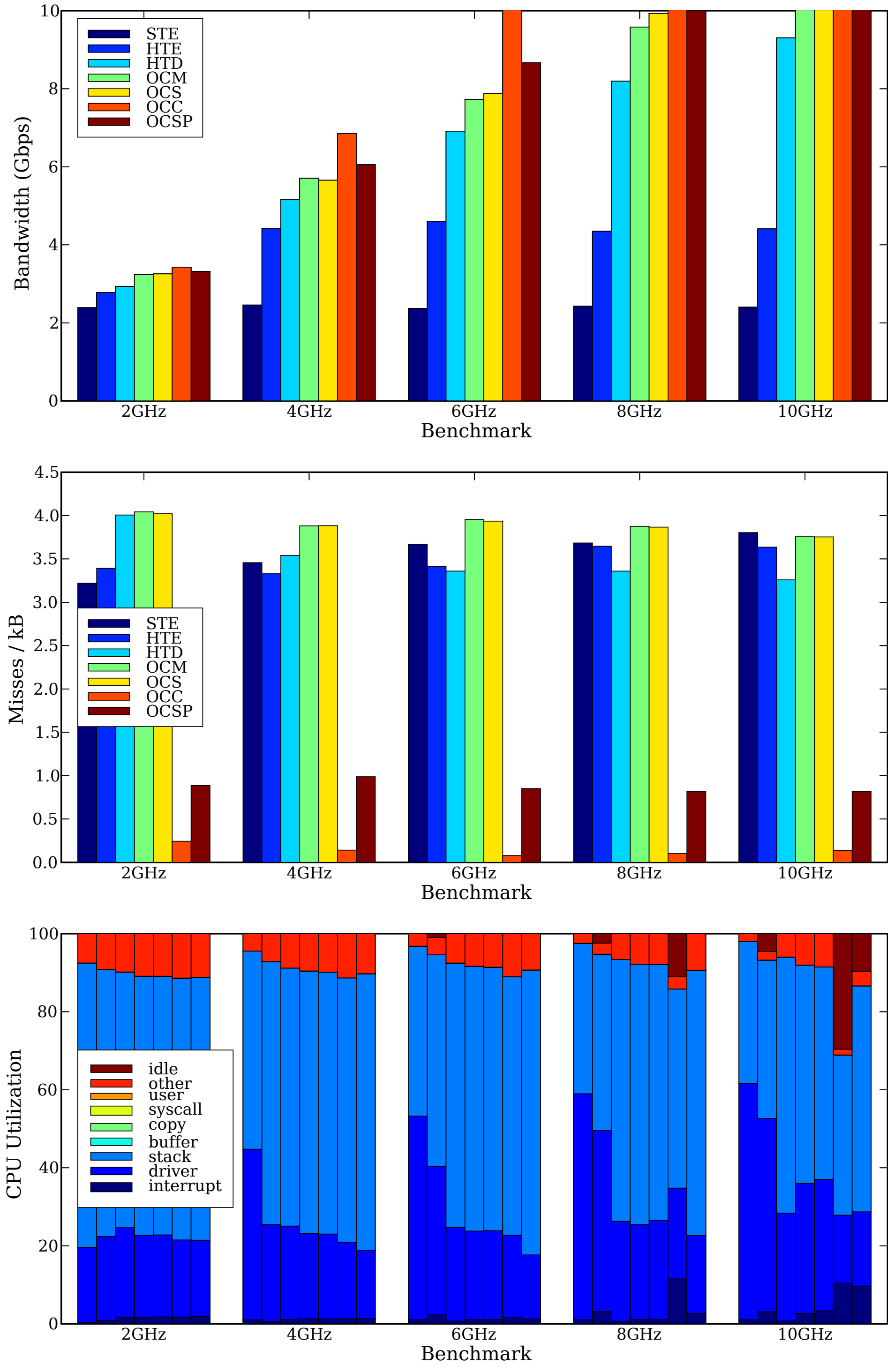


Figure 5.9: Network Address Translation Gateway Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2 GHz to 10 GHz with a 4 MB level 2 cache size.

sively, its performance sees the greatest benefit from the combination of the on-chip NIC and in-cache data placement.

The misses/KB graph in Figure 5.9 shows that the OCC configuration eliminates nearly all misses, while OCSP eliminates misses on only header data, as expected. For the 8 GHz and 10 GHz CPU configuration, OCC and OCS are able to saturate the network link with CPU cycles to spare.

5.6 Conclusions

This chapter investigated the performance impact of tighter coupling between a 10 Gbps Ethernet NIC and the CPU. Overall, tighter integration is clearly a performance win in all cases. The results show the clear benefit of an on-die NIC in terms of bandwidth and CPU utilization, particularly on the transmit side. Moving the NIC onto the CPU die itself provides a major opportunity for closer interaction with the on-chip memory hierarchy. Here the results also show the potentially 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. As far as these benchmarks show, the DMA of packet data directly to the cache is always a good idea: it can, in some circumstances, yield huge performance differences, while not adversely affecting performance or cache miss rate in the others.

As stated above, I have shown that 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 general, larger caches and/or faster CPUs increase the likelihood that the processor will touch network data before it is evicted from the cache. In contrast, smaller caches and slower CPUs are less likely to do so, and thus do not benefit from cache placement of DMA data. If comparing this result to previous work I presented [8], a clear discrepancy can be seen. The discrepancy is specifically with the number of cache misses in the OCC case. I initially thought that the high cache miss rate for OCC in applications such SPEC WEB99 was due to pollution. It turns out that the high miss rate was due to the allocation of L2 cache blocks on DMA reads. For example, when transmitting a packet, if the data was in main memory, it would be processed as any other cache miss resulting in a block being evicted and the new data being brought in. Since DMA data generally has a low likelihood of being touched once read, this policy is a bad choice and it caused an inflated cache miss rate for those transmits that were not already in the L2 cache. I corrected this in these experiments by not allocating blocks in the cache for DMA reads that miss in the cache. The header splitting configuration was a first step in attempting to mitigate cache pollution while still achieving some of the benefit. While no benefit was shown in this work, in systems with a very small cache, or applications with a cache footprint that just fits in the cache, it is likely that a benefit could be seen.

Chapter 6

The Simple Integrated Network Interface Controller

Although users can now plug 10 Gbps Ethernet links into their server systems, getting those systems to keep up with the bandwidth on those links is challenging. One proposed solution is to perform much of the required protocol processing on an intelligent NIC, referred to as a TCP offload engine (TOE) (See Chapter 8). This offload approach reduces host CPU utilization and, perhaps more importantly, reduces the frequency of costly interactions between the CPU and NIC across the relatively high-latency I/O bus. Due to the high transfer rate and the complexity of the TCP protocol, a 10 Gbps TOE requires a substantial amount of general-purpose processing power.

This trend of pushing more intelligence out to the NIC brings to mind Myer and Sutherland's "wheel of reincarnation" [59]. They observe that a peripheral design tends to accrue more and more complexity until it incorporates general-purpose processing capabilities, at which point it is replaced by a general-purpose processor and a simple peripheral, corresponding to a full turn of the wheel. Providing computational resources via an additional symmetric CPU imparts more flexibility and generality than using a dedicated, asymmetric, special-purpose processing unit.

This chapter, in which I propose a system architecture for 10GigE servers, represents the completion of a turn around the wheel for NIC design. I argued in Chapter 5 that integration of the NIC onto the CPU die should be considered because high-bandwidth TCP/IP networking places a significant burden on end hosts. Integration of the NIC onto the CPU die changes the fundamental design constraints presented to the NIC architect. These constraints change primarily due to the different nature of communication between the CPU and NIC. As discussed in Chapter 5, with an

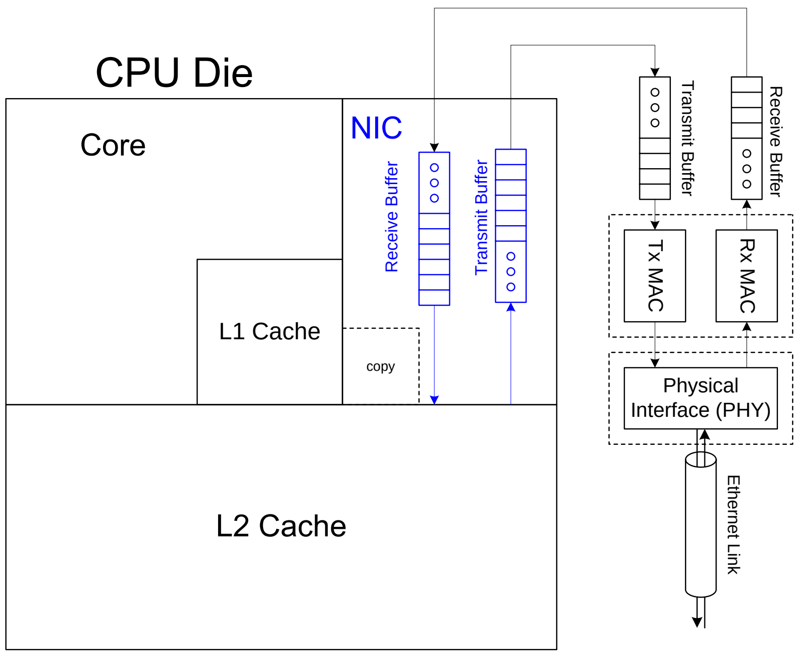


Figure 6.1: SINIC Block Diagram - This design integrates a checksum/copy unit and the FIFOs onto the CPU die, leaving the physical interfacing components off chip. While this figure depicts the entire FIFOs on the CPU die, this organization is not the only one possible. One can imagine putting a significant amount of buffering off-chip, leaving only a minimal storage on chip.

integrated NIC, the communication latency is reduced by orders of magnitude and the bandwidth is potentially increased by orders of magnitude. Additionally, the NIC can more easily communicate with other on-chip structures that might be difficult to access otherwise (e.g. caches, TLBs). This chapter argues that, given this potential, latency can be reduced more effectively by integrating a simple NIC with the host CPUs than by pushing additional computation out to the NICs. This simple integrated NIC (SINIC) is not specifically designed to have a higher performance than would be found with the integration of a traditional NIC; rather, its design provides comparable performance with added flexibility and reduced implementation complexity.

With SINIC (see Figure 6.1), I strip the NIC down to its most basic components—a pair of FIFOs—supplemented only by a block copy/checksum unit. All other processing is performed by the device driver on a general-purpose host CPU. The driver directly initiates individual transfers between the FIFOs and main memory—a task typically performed by a DMA engine in even the most basic NICs, but which requires non-trivial computational resources at 10 Gbps [85].

Using detailed full-system simulation, I show that SINIC provides performance comparable to a conventional DMA-descriptor-based NIC design when the latter is also integrated onto the processor die (as shown in the last chapter). Even though SINIC does not provide asynchronous DMA, it is capable of providing significantly higher performance than a DMA-based off-chip NIC because of its lower CPU access latency and its ability to place incoming DMA data into the on-chip cache.

6.1 The Case for a Simple Network Interface

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

A high-bandwidth NIC requires significant amounts of closely coupled processing power in any design. The key enabler for my SINIC approach is the ability to have one or more general-purpose host CPUs provide that power. This coupling is easily achieved by integrating the NIC on the same die as the host CPU(s).

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

In the previous chapter, I discussed the case for integrating the NIC on a processor die, a prerequisite for the SINIC structure presented in this chapter. In the rest of this section I contrast my approach with two alternatives: current conventional NIC designs and the TOE approach that represents a contrasting vision of future NIC evolution.

6.1.1 Simple Versus Conventional NICs

The interface to a conventional Ethernet NIC also consists of a pair of FIFOs plus control and status information. The control and status information is typically exchanged with the CPU through uncached, memory-mapped device registers. A conventional NIC resides on a standard I/O bus (e.g., PCI) that is physically distant from and clocked much more slowly than the CPU, so these uncached accesses may require thousands of CPU cycles to complete (Section 4.3). Providing access to the network data FIFOs via these memory-mapped device registers is impractically slow, so the NIC uses DMA to extend its hardware FIFOs into system memory.

To give the operating system some flexibility in memory allocation, the memory-resident FIFOs are divided into multiple non-contiguous buffers. The address and length of each buffer is recorded in a DMA descriptor data structure, also located in main memory. The transmit and receive FIFOs are represented as lists of DMA descriptors (Figure 2.3).

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

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

Though this process of fetching, processing, and updating DMA descriptors is conceptually simple, it incurs a non-trivial amount of memory bandwidth and processing overhead, both on the NIC and in the device driver. Willmann et al. [85] analyze a commercial 1 Gbps Ethernet NIC that implements this style of DMA queue in firmware, and determined that an equivalent 10 Gbps NIC must sustain 435 MIPS to perform these tasks at line rate, assuming full 1518-byte frames. They proceed to show how this throughput can be provided in a power-efficient way using a dedicated six-way 166 MHz embedded multiprocessor. Note that, other than possibly calculating checksums, this computational effort provides no inspection or processing of the packets whatsoever; it merely serves to extend the network FIFOs into system memory where abundant buffering can hide the high latency of CPU/NIC communication. Although these functions could be implemented more efficiently in an ASIC than in firmware, this approach bears the traditional ASIC burdens of cost, complexity, and time to market for no additional functional benefit.

In contrast, SINIC, with its direct exposure of the hardware FIFOs to software, does not require DMA descriptors at all, avoiding the management overhead. On transmit, the device driver merely copies the packet from the data buffer(s) into the FIFO. As soon as the copy completes, the buffers are free to be reused. On receive, the host CPU is notified via an interrupt if necessary, and the driver copies data from the receive FIFO into a buffer. My SINIC design, described in detail in Section 6.2, includes a simple block copy engine to make these copies more efficient, but little else. The reduced latency afforded by on-chip integration allows the NIC to operate without the expanded buffer space provided by the DMA descriptor queues.

6.1.2 SINIC Versus TOE

Rather than integrating the NIC near the CPU, systems can closely couple processing with the NIC by leaving the NIC on an I/O bus and adding processing power to it. An extreme example of this approach is a TOE, in which the NIC itself is responsible for most or all of the TCP and IP protocol processing [2, 11].

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

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

A corollary of this lack of flexibility is that TOEs are not easily customized for or tightly integrated with particular operating systems. The fact that the code controlling copies out of the SINIC receive FIFO is part of the device driver, and thus has immediate access to the full kernel code and data structures, is critical to achieving my zero-copy extensions in Section 7.2.

Other arguments against this direction include the inability of TOEs to track technology-driven performance improvements as easily as host CPUs [57, 71] and the fact that TOEs provide significant speedups only under a limited set of workload conditions [77].

6.2 The Simple Integrated Network Interface Controller (SINIC)

This section describes the detailed design of one possible low-level NIC interface—my simple integrated NIC (SINIC)—and its associated device driver. SINIC by itself is not intended to provide higher performance than a similarly integrated conventional NIC. Instead, its design provides comparable performance with added flexibility and reduced implementation complexity.

6.2.1 SINIC Design

As discussed in Section 6.1, conventional NICs provide a software interface that supports the queuing of multiple receive and transmit buffers via a DMA descriptor queue. Due to its close proximity to the host CPUs, SINIC is able to achieve comparable performance without a queuing interface and without scatter/gather DMA.

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

SINIC operates entirely on physical addresses. Because it is designed for kernel-based TCP/IP processing, it does not face the address translation and protection issues of user-level network interfaces.

The RxDone and TxDone registers provide status information on their respective FIFOs. Each register indicates the number of packets in the FIFO, whether the associated copy engine is busy, whether the last copy operation initiated on the FIFO completed successfully, and the actual number of bytes copied.[1] TxDone also indicates whether the transmit FIFO is full. RxDone includes several additional bits. One bit indicates whether there is more data from the current packet in the FIFO. Another set of bits indicates whether the incoming packet is an IP, UDP, or TCP packet, and whether SINIC's calculated checksum matched the received packet checksum.

SINIC implements a single copy engine per FIFO.[2] As a result, the CPU must wait for the previous copy to complete before initiating another copy. Because individual buffer transfers are relatively fast, the driver simply busy-waits when it needs to perform multiple copies. SINIC enables more efficient synchronization through two additional status registers, RxWait and TxWait. These registers return the same status information as RxDone and TxDone, respectively, but a load to either of these registers is not satisfied by SINIC until the corresponding copy engine is free. Thus a single load to RxWait replaces a busy-wait loop of loads to RxDone, reducing memory bandwidth and power consumption.

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

Like a conventional NIC, but unlike a TOE, SINIC interfaces with the kernel through the standard network driver layer. SINIC's device driver is simpler than conventional NIC drivers because it need not deal with allocation of DMA descriptors, manage descriptors and buffers (e.g., reclaim completed transmit buffers), nor translate the kernel's buffer semantics (e.g., Linux `sk_buffs`) into the NIC's DMA descriptor format. With SINIC, there are no descriptors. When a packet must be transmitted, the device driver simply loops over each portion of the packet buffer initiating a transmit with a programmed I/O (PIO) write to TxData, busy-waiting on the result with a PIO read to TxWait. For the final portion of the packet, the driver does not wait on the copy to complete; instead, it allows the copy to overlap

[1]This last value is useful as it allows the driver to provide the allocated buffer size as the copy length to the receive FIFO and rely on SINIC to copy out only a single packet even if the packet is shorter than the buffer.

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

with computation, and verifies the engine to be free before initiating the next packet transmission.

6.2.2 The SINIC Device Driver

One major benefit of SINIC is the ability to use it without major modifications to the operating system kernel. The only software necessary is a device driver, something required by all NICs. In fact, this device driver can be significantly simpler than other device drivers because it is no longer necessary to translate the kernel's buffering semantics to the NIC's buffering semantics and deal with the resource allocation of the NIC buffers. For example, on the transmit of a single packet, the device driver for a conventional NIC must allocate one transmit descriptor for each separate portion of the packet buffer; the driver must then copy the buffer information from the kernel buffers to the transmit descriptors; next, the driver will signal to the device that something is ready for transmit with a programmed I/O; finally, when the transmit it complete, the device will signal the driver using an interrupt at which time the device will deallocate the buffer descriptor so it may be reused. With SINIC, there are no descriptors. When a packet must be transmitted, the device driver simply loops over each portion of the packet buffer initiating a transmit with a programmed I/O write, and busy waiting on the result with a programmed I/O read.

6.3 Results

In this section SINIC with direct attachment to the on-chip last level cache is compared to a conventional NIC with various points of attachment. First, to set a baseline, measurements are taken with a conventional NIC attached to the CPU die via a HyperTransport like bus and a PCI Express bus. Next, two on-chip configurations of a conventional NIC are provided, including the following: (1) attachment on the far-side of the last level of on-chip cache, giving it DMA access to the on-chip memory controller, but not the cache; and (2) attachment to the near-side of the last level of on-chip cache, giving it access to the last level cache.

Figure 6.2 shows the aforementioned NICs and attachments running the Netperf micro-benchmarks. Looking at the Commodity NIC (CNIC) attachments it is clear that tighter integration of the NIC in the system positively increases performance showing a better than 60% improvement in all cases. There are two main reasons for this: the device can be accessed with a much lower latency and the device can place

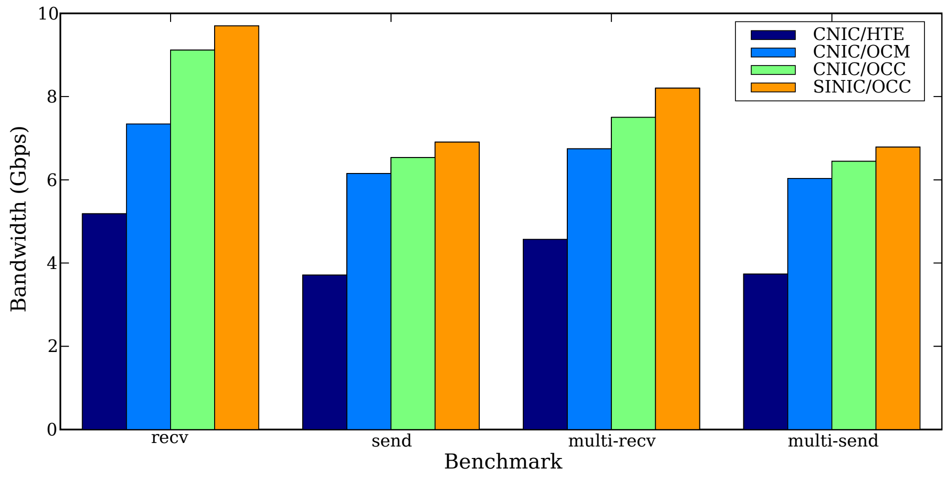


Figure 6.2: Micro-Benchmark Bandwidth - Measurements of achieved bandwidth of the various micro-benchmarks with CPUs running at 4GHz.

data directly into the last level of cache, thus reducing the amount of memory traffic and cache misses. Placing data directly in the cache can reduce the number of cache misses per kilobyte of data transmitted by as much as 15x [8, 38].

The same graph shows that SINIC attached to the near side of the last level cache outperforms the more complex CNIC attached at the same position. SINIC doesn't have the overhead that a normal NIC has and even though the data must be read with programmed I/O the device is so close to the CPU that this is faster than writing the data to memory and managing the buffer structures necessary to accomplish this task.

Figure 6.3 depicts the cache behavior of the micro-benchmarks in terms of the number of cache misses per kilobyte of data transferred. The results are very similar to those shown in the previous chapter: the receive-oriented benchmarks which DMA to the memory miss very frequently in the cache, the receive-oriented benchmarks which DMA to the cache miss very infrequently, and the transmit-oriented benchmarks miss infrequently in both DMA cases.

Figure 6.4 shows a breakdown of where the CPU spent time. Note the reduction seen between the PCI Express attached NIC and the other attachments. Since the device is closer, accessing its device registers is a much cheaper operation which provides more available processing time for the TCP/IP stack.

Looking at more complex macro-benchmarks, one may worry that having to manually coordinate each copy from/to the SINIC FIFOs into/out-of the kernel would provide an unacceptable overhead compared to a DMA engine doing the work with-

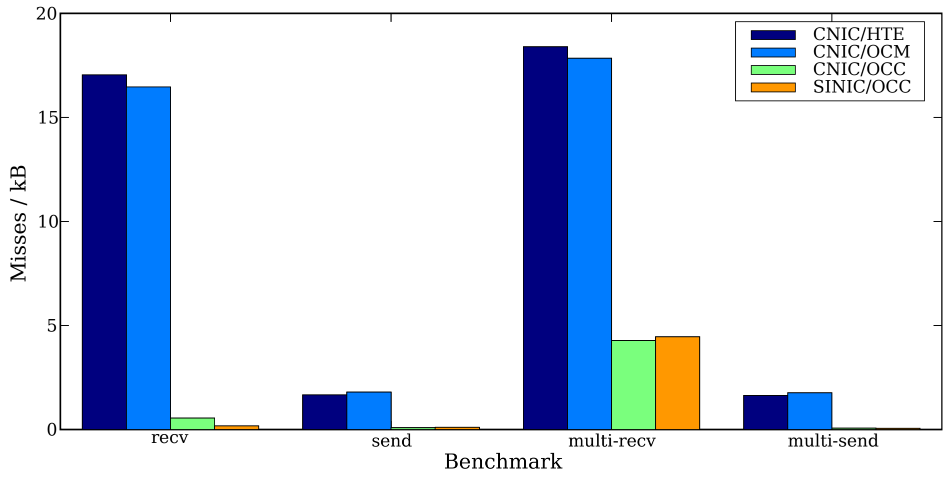


Figure 6.3: Micro-Benchmark Cache Performance - Cache miss rates represented in misses per kilobyte transferred for the various micro-benchmarks with CPUs running at 4GHz.

out direct CPU interaction. However, as can be seen from Figure 6.5, this is not the case. Even for more complex workloads including CPU bound ones like SPECWeb99, no decrease in performance is observed.

6.4 Conclusion

I have described a simple network interface—SINIC—designed to be integrated onto the processor die to support high-bandwidth TCP/IP networking. SINIC is simpler than conventional NICs in that it avoids the overhead and complexity of DMA descriptor management, instead exposing a raw FIFO interface to the device driver. Despite its simplicity, detailed full-system simulation results show that SINIC performs as well as or better than a conventional NIC given the same level of integration.

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

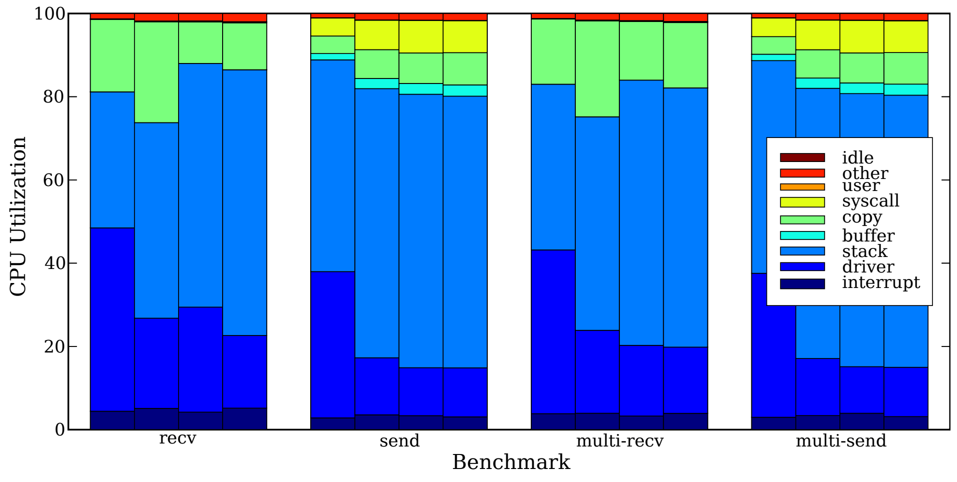


Figure 6.4: Micro-Benchmark CPU Utilization Breakdown - Breakdown of CPU utilization for the various micro-benchmarks with CPUs running at 4GHz.

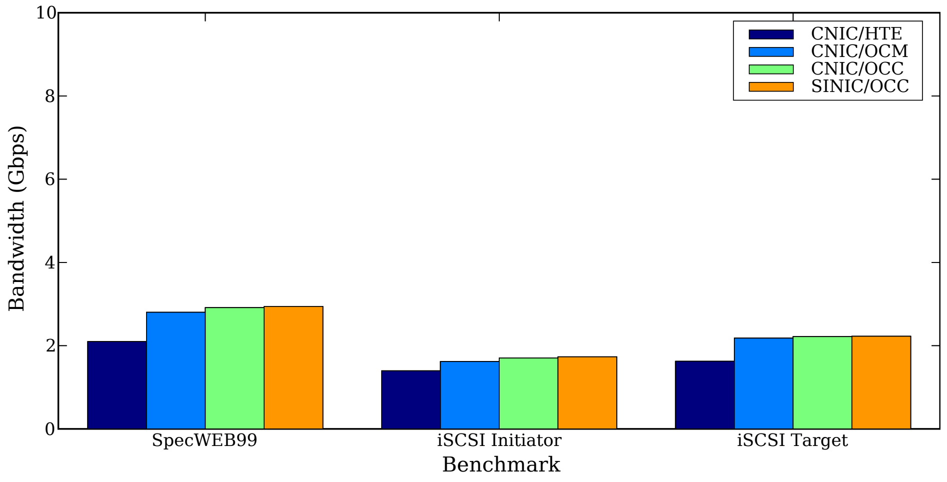


Figure 6.5: Macro-Benchmark Bandwidth - Measurements of achieved bandwidth in gigabits per second of the various macro-benchmarks. The measurements are for the conventional NIC in the HTE, OCM, OCC, and OCS configurations as described in Figure 5.2 and SINIC OCC, all running at 4GHz with an 8MB level 2 cache.

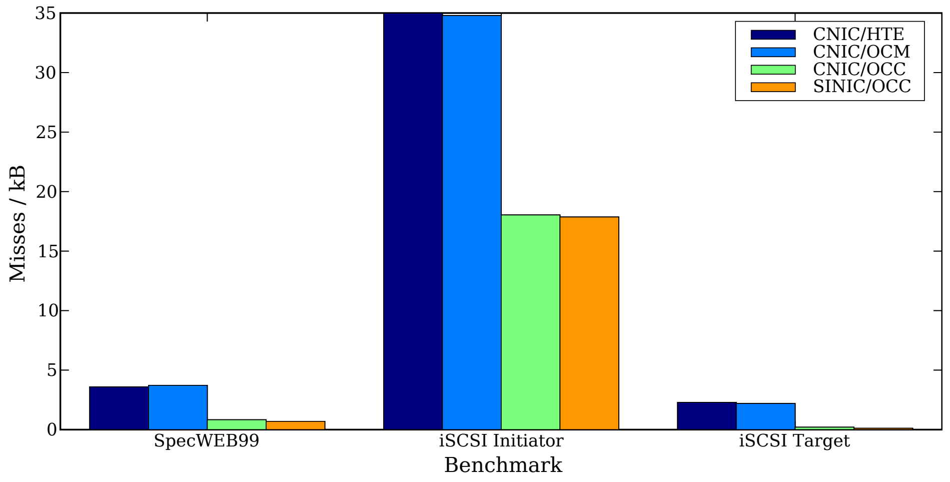


Figure 6.6: Macro-Benchmark Cache Performance - Measurements of the cache miss rate measured in misses per kilobyte of data transferred for the various macro-benchmarks. The measurements are for the conventional NIC in the HTE, OCM, OCC, and OCS configurations as described in Figure 5.2 and SINIC OCC, all running at 4GHz with an 8MB level 2 cache.

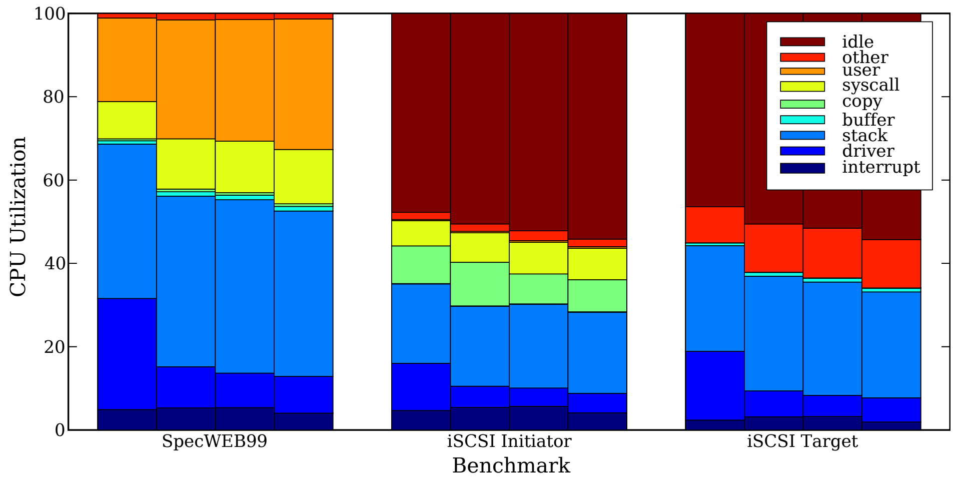


Figure 6.7: Macro-Benchmark CPU Utilization Breakdown - Measurements of CPU utilization broken down into several categories for the various macro-benchmarks. The measurements are for the conventional NIC in the HTE, OCM, OCC, and OCS configurations as described in Figure 5.2 and SINIC OCC, all running at 4GHz with an 8MB level 2 cache.

Chapter 7

Virtualized SINIC and Zero-Copy

In Chapter 6, I described SINIC, a simplified NIC designed for integration onto a CPU die. A key potential benefit of exposing the NIC FIFOs to host CPUs, as SINIC does, is that kernel software can control the memory destination of each packet explicitly. SINIC's flexibility is limited only by its adherence to a strict FIFO model which in turn limits the amount of packet-level parallelism the kernel can exploit. This limitation is overcome by V-SINIC, an extended version of SINIC that provides virtual per-packet registers, enabling packet-level parallel processing while maintaining a FIFO model. To illustrate the flexibility added by virtualizing SINIC, I describe a modest set of Linux kernel extensions that allow the protocol stack to defer copying payload data out of the receive FIFO until the packet's protocol header has been processed. If the receiving user process has posted a receive buffer (e.g., by calling `read()`) before the packet's arrival, the packet payload can be copied directly from the NIC FIFO to the user's buffer, achieving true "zero copy" operation in the Linux 2.6 kernel.

Because the base SINIC design presents a plain FIFO model to software, each packet must be copied out of the FIFO before the following packet can be examined. This strict FIFO restriction significantly limits packet-level parallelism when the deferred copy technique is applied. I remove this restriction by adding virtual per-packet control registers to the SINIC model. This extended interface enables overlapped packet processing in both the transmit and receive FIFOs, including lockless parallel access to the FIFOs on chip multiprocessors.

SINIC, while unable to process packets in parallel, is able to dynamically select the data buffer used when a packet is received. This is unlike the DMA descriptor model where receive buffers must be populated by the driver in advance.[1] In Section 7.2, I describe a set of kernel modifications that take advantage of V-SINIC's ability to process packets in parallel and dynamically select data buffers. These modifications provide true zero-copy receives—where data is copied directly from the NIC FIFO into the user's destination buffer—for unmodified socket-based applications.

7.1 V-SINIC for Packet-Level Parallelism

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

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

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

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

payload). This policy applies to the transmit FIFO only; as will be described shortly, the receive FIFO is specifically designed to allow interleaving of headers and payloads from different packets as they are copied out.

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

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

7.2 Implementing Zero-Copy Receives on V-SINIC

The overhead of copying packet data between kernel and user buffers is often a significant bottleneck in network-intensive applications. This overhead can be avoided on the receive side by copying data directly from the NIC FIFO into the user buffer. Unfortunately, this "zero-copy" behavior is practically impossible to achieve with a conventional DMA-descriptor-based NIC, as receive packet buffers must be posted to the NIC before the driver has any knowledge of which connections the arriving packets will be destined.

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

The zero-copy scheme does address translation in software before the copy from kernel to user space is done and because this code is in the kernel, there are no security issues to deal with. The data is copied directly from the NIC FIFO to the user-space buffer. This particular copy is performed synchronously by the CPU, so I believe that pro-active page pinning is unnecessary because the copying CPU will not respond to a TLB shootdown request until the copy is complete. However, it would be trivial to set a busy bit on the page before the copy began to indicate to the pager on other processors that the page is unavailable for swapout.

7.3 Results

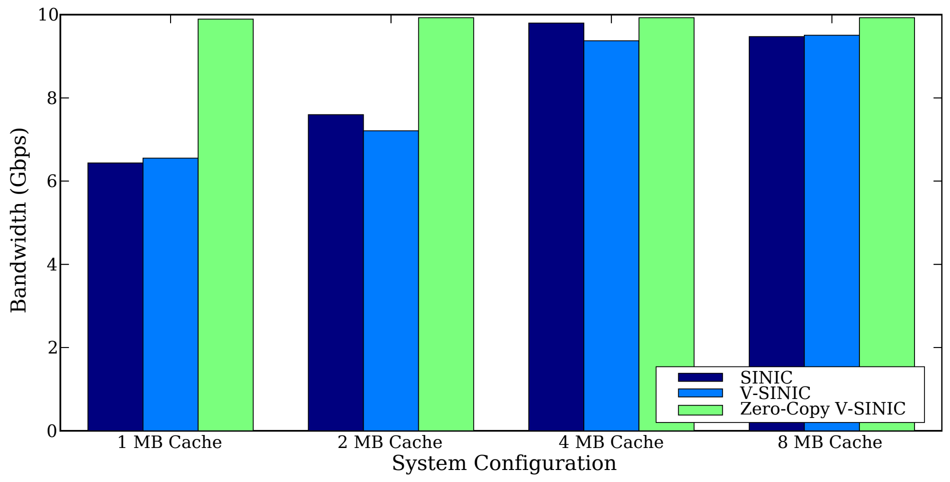


Figure 7.1: Receive Zero-Copy Bandwidth - Achieved bandwidth for the receive micro-benchmark with CPUs running at 4GHz.

I show the overall performance results of the zero-copy optimization in Figure 7.1. SINIC is able to saturate the network with a large (4-8 MB) L2 cache, because its cache placement of incoming network data makes the buffer copies efficient cache-to-cache operations. However, for smaller L2 cache sizes, the SINIC network buffers overflow into main memory. The overhead of the resulting DRAM-to-cache buffer copies causes significant bandwidth degradation (a 40% reduction down to 6 Gbps with a 1 MB cache). Without the zero-copy optimization, V-SINIC provides similar performance characteristics, though at a slightly reduced performance level due to additional overheads in the device driver.[2] In contrast, the zero-copy optimization

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

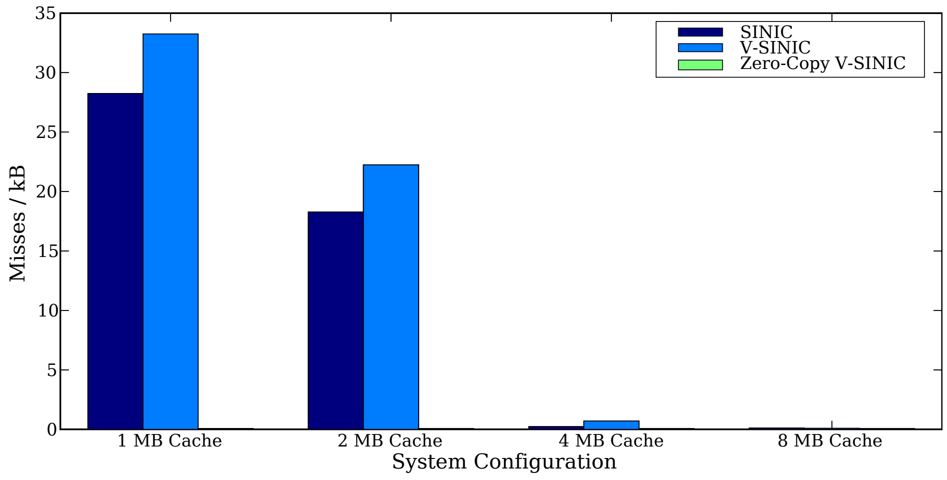


Figure 7.2: Receive Zero-Copy Cache Miss Rates - Cache miss rate in misses per kilobyte of data transferred for the receive micro-benchmark with CPUs running at 4GHz.

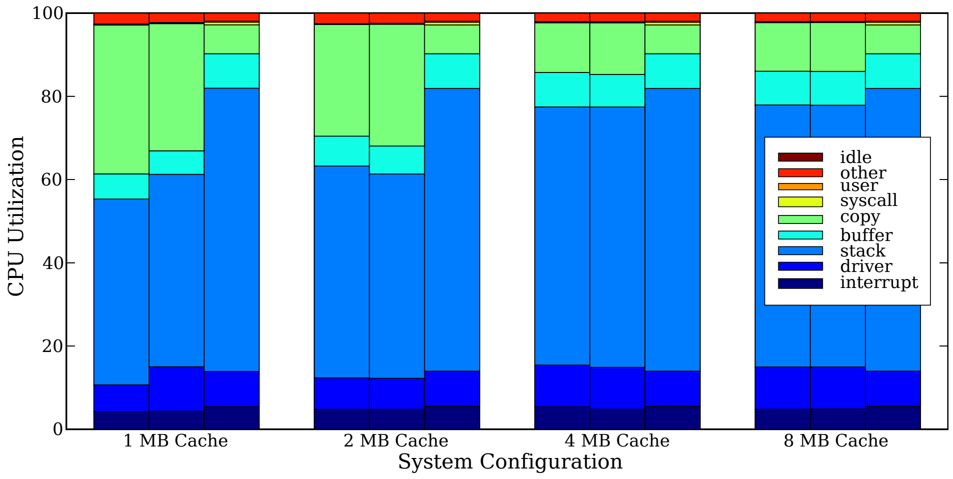


Figure 7.3: Receive Zero-Copy CPU Utilization Breakdown - Breakdown of CPU utilization for the receive micro-benchmark with CPUs running at 4GHz.

eliminates the buffer copy entirely, making performance insensitive to the cache size, and allowing the system to saturate the network in every configuration. Figure 7.2 shows that the number of cache misses is strongly correlated to network performance. While SINIC's cache data placement coupled with a sufficiently large cache drives the cache miss rate of the buffer copies to zero, the zero-copy V-SINIC model incurs practically no cache misses because network data is read directly from the FIFO rather than from the memory system. Figure 7.3 shows that, as a result of dramatically reduced copy time, the zero-copy V-SINIC system can spend more time in the TCP/IP stack processing packets. The copy time shown for the zero-copy V-SINIC case corresponds to the time the CPU spends setting up and waiting on the V-SINIC copy engine to copy data from the FIFO to the user buffer.

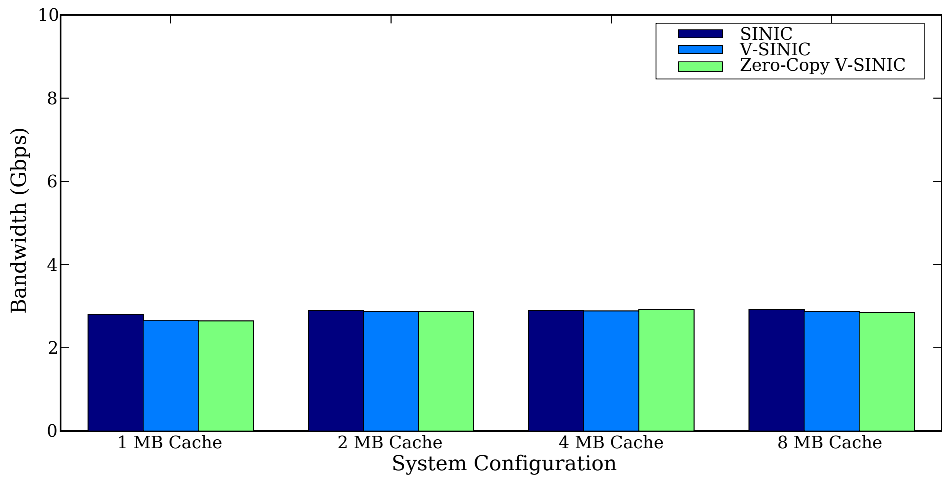


Figure 7.4: SpecWEB Zero-Copy Bandwidth - Achieved bandwidth for the web benchmark with CPUs running at 4GHz.

Figure 7.4, Figure 7.5, and Figure 7.6 show the zero-copy results for SpecWEB. In this case, we see that zero-copy has only a relatively minor impact on performance compared to normal SINIC. This is simply because SpecWEB is a transmit oriented benchmark and does not benefit from zero-copy on receive. Similar results for the transmit micro-benchmark, and for the iSCSI benchmark can be found in Appendix B.

It is possible for zero-copy to significantly degrade the performance of some benchmarks if it is always active. This occurs when the buffering in the NIC FIFO is too small compared to the number of outstanding requests needed by the workload. In order to not degrade the performance of SpecWEB, it was necessary to tune the system to be less aggressive with zero-copy. In this mode, the system detects that the FIFO is filling and disables zero-copy to avoid dropping packets. While the backoff

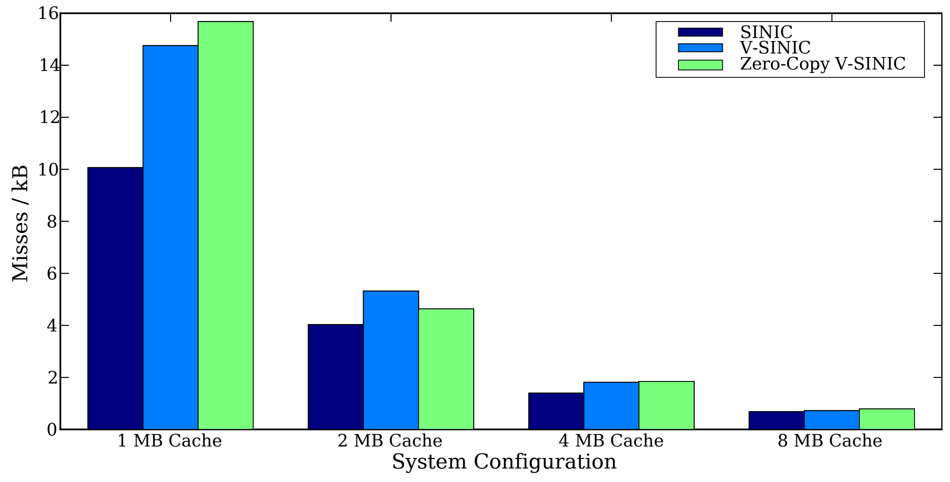


Figure 7.5: SpecWEB Zero-Copy Cache Miss Rates - Cache miss rate in misses per kilobyte of data transferred for the web benchmark with CPUs running at 4GHz.

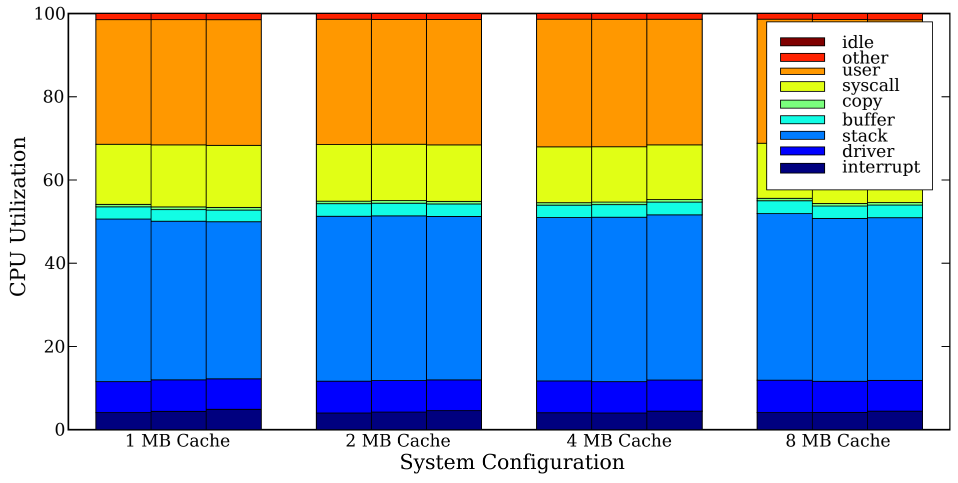


Figure 7.6: SpecWEB Zero-Copy CPU Utilization Breakdown - Breakdown of CPU utilization for the web benchmark with CPUs running at 4GHz.

response to the FIFO fill is automatic, it did reduce the zero-copy performance for the receive benchmark by about 15%. The iSCSI and TCP transmit benchmarks were unaffected either way. Thus the results shown for all of the benchmarks except for SpecWEB were run without the detection, while the results for SpecWEB have the detection and backoff enabled. Improving the detection so that it is fully automatic is future work.

7.4 Conclusion

In this chapter, I presented a novel approach to extending SINIC's FIFO-based interface to allow packet-level parallelism both on transmit and receive. By associating a set of "virtual" FIFO registers to each packet, the V-SINIC interface allows lockless concurrent packet transmission on multiprocessors and enhanced parallelism in receive packet processing. V-SINIC also enables a deferred-copy technique that supports a straightforward implementation of zero-copy receive handling, which I have implemented in the Linux 2.6 kernel.

Chapter 8

Related work

There has been a great deal of work in the area of improving networking performance relating directly to reducing the overheads associated with high-speed networking at the end host. In this chapter, I group the related work into sections that have a common theme, starting with primarily hardware-oriented schemes and moving to the primarily software-oriented schemes. Much of the work in the can fit in multiple logical categories, but is discussed in the first section in which it belongs.

8.1 Hardware Offload

Hardware offload refers to solutions that take pieces of the protocol stack and move them to the network interface controller. The intent of offloading is that by removing the responsibility for a particular task from the CPU, the CPU overhead of networking is reduced. Because NIC vendors cannot redesign any other part of the system, offloading is the primary mechanism through which they can provide performance improvements to their customers.

Offloaded tasks span the range from the very simple, such as checksum calculation, to the very ambitious, such as the entire TCP protocol. The difficulty presented by offload is that part of the protocol stack—usually something embedded somewhere in the middle—is moved out of the kernel to the NIC device, violating the normal layering of protocol processing. The kernel must be adapted to omit this processing and defer it to the device, typically requiring specific and sometimes awkward changes to the operating system protocol stack. This violation of layering also often means that opportunity for offload is lost if the user does not use the protocol stack in the specific way that the NIC expects.

8.1.1 Hardware Address Filtering

One of the very first examples of offloading was the addition of packet filtering to the network interface controller. In the early days of Ethernet, the medium was a shared bus, and a host would receive many packets destined for other hosts. Without a hardware filter, these packets would be copied completely into main memory, only to discover after reading the 14-byte Ethernet header that many packets were useless. This task was simple in the early days when the only addresses that were of concern were the local address and the broadcast address. As multicast became more commonly employed, it became necessary for a host to be able to listen to a potentially large number of addresses. This has led to many adapters supporting a larger programmable filter that can be updated on-the-fly. Though switched networks have replaced the shared-bus medium today, Ethernet adapters still support filtering.

8.1.2 Checksum Offload

The checksum calculations in IP, TCP, and UDP are well known for their performance cost in high speed networking [18, 45, 30]. Their simplicity makes them a prime target for hardware offload. IP, TCP, and UDP all have a simple checksum that must be calculated on every packet. (The IP checksum is over the header only, while TCP and UDP checksums cover the respective protocol headers and the payload.) For all three protocols, the checksum is the 16-bit one's complement sum of the protected words. The calculation is not computationally expensive like a CRC-32, but the fact that the CPU must touch all of the words—which are likely in main memory—makes it a time-consuming operation. Unlike address filtering, where no protocol software changes are necessary,[1] checksum offload requires the checksum calculation to be moved out of the middle of the protocol stack to the device. This requires that the kernel protocol stack be modified to allow the option of not calculating the checksum. Even though the checksum calculation is a simple operation, it was many years after the introduction of the feature before operating systems were modified to use it as a general feature. As with other offload techniques, checksum offload suffers from the inability to cope with protocols other than basic IP, TCP, and UDP. If, for example, traffic is encapsulated by IPsec, the TCP or UDP packet is encrypted and encapsulated within IPsec header. Because the data is encrypted

[1]The filtering logic must still remain in the operating system kernel since most packet filters cannot capture everything.

before it reaches the NIC, the NIC does not have access to the raw data to calculate the correct checksum and insert it in the packet.

8.1.3 TCP Segmentation Offload (TSO)

TCP Segmentation Offload allows the protocol stack to send the NIC a TCP segment larger than the maximum transmission unit (MTU). The NIC will then break the single large packet into multiple TCP segments before sending it onto the Ethernet.[2] In TSO, the NIC reuses the original header on each of the new segments, fixing up the sequence number and checksum fields to reflect the changes. Since the checksum must be calculated for each new segment, this optimization must be used in conjunction with checksum offload. The TSO optimization reduces the per-packet costs in the software protocol stack by requiring fewer traversals through the transmit path.

8.1.4 TCP Offload Engines (TOEs)

As mentioned above, much of the effort in enhancing commercial NICs has focused on offloading work from the CPU to the NIC. Minor tasks such as checksum generation and transmit segmentation are common in current Gigabit Ethernet NICs. TCP offload engines (TOEs) [2] take the offload concept to its logical conclusion, moving all TCP/IP processing onto the NIC. TOEs address the high cost of CPU/NIC interactions by having the CPU interact with the NIC less frequently at a higher semantic level. SINIC takes an alternate approach by directly attacking the NIC/CPU communication overhead.

Although TOEs can greatly reduce the CPU overhead of protocol processing, there are a number of arguments against the TOE approach. Perhaps the most critical are the inability of a TOE to stay on the general-purpose CPU performance curve and the difficulty in dealing with bugs in firmware-based TCP code [57].

Another major problem with the TOE solution is that the TOE device must implement every protocol combination that the kernel might want to use. A Microsoft document on Chimney [55], the recently introduced Windows kernel device driver

[2]In theory, the NIC could create IP fragments instead of TCP segments. However, TCP segmentation is the expected method for dealing with large transmissions. Fragmented IP packets do not follow the fast path through the kernel protocol stack.

interface for protocol offload engines, states that twenty-four RFCs[3] are of "strong interest to offload targets," including TCP, UDP, IP version 4 (IPv4), and IPv6; another six a designer should "consider supporting;" and thirty-six more, including IPSec, that are "of interest." This large number of protocols represents untold man-years of development effort. In the Linux kernel, for example, there are approximately 100,000 lines of actual source code in the core, IPv4, and IPv6 source directories. It would be necessary for someone to either write all of this code from scratch for the NIC firmware, or port some existing protocol stack to this new platform.

This need for both the kernel and the device to know about all of the protocol combinations desired is a slippery slope that can easily impede future protocol design. Protocols like IPSec and IP-in-IP encapsulation and applications like packet filtering are examples of situations where TOEs have problems. One could, for example, implement IPSec on the TOE NIC, but any further change to the protocols would also have to be implemented in the TOE to get any benefit. Recently, a new IETF standard was proposed for the encapsulation of IPSec packets within UDP frames [39]. If a user had a TOE that supported IPSec, that TOE would be useless without an upgrade if the user wished to use this new feature.

Packet filtering, used by firewalls to enforce network usage policies or protect systems from undesired or malicious traffic, is an even larger challenge for TOEs.[4] The TOE must implement the entire packet filter or it will be useless and reduced to behaving like a normal NIC. This all-or-nothing requirement is due to the fact that when a host desires to filter packets, it compares each packet to a specific set of rules (which for the most part use IP and TCP header information) to determine if the packet is accepted. In a TOE design the IP and TCP header information is not provided to software, making it impossible for the operating system to make these choices. The problem with putting the filter on the TOE is that many filter rules are designed to be stateful. In a large server or firewall, the amount of state kept for the filters can be huge. However, a TOE already has to record a lot of connection state information to manage all of the TCP connections it terminates, so it may not require a great deal of extra state to implement the filter on the TOE. Unfortunately,

[3]An RFC (Request For Comment) is the informal mechanism that is used to create standards on the Internet. Each protocol revision is assigned a unique RFC number (e.g. IP (RFC 791), TCP (RFC 793), and UDP (RFC 768)). There are currently over 4,300 RFCs published.

[4]A firewall may either be a separate system inserted in the network at the ingress point, or software residing on the host that is interposed somewhere between socket interface and the device driver.

this still leaves the problem that the OS and the NIC would then have to agree on filter rules and changing the capability of the filter would be difficult.

Intel claims that TOEs by themselves do not provide significant performance improvements because they do not address the overhead of data movement between the NIC and main memory. As a result, they have recently begun a platform-level attack on I/O performance with the announcement of an "I/O Acceleration Technology" (I/OAT) initiative [50] that explicitly discounts TOEs in favor of a "platform solution" [32]. I/OAT includes several optimizations mentioned elsewhere in this chapter, including header splitting, a hardware copy engine, interrupt moderation, and software protocol stack optimizations.

8.2 Interface Improvement

Some previous work has specifically addressed the fact that the latency and bandwidth between the CPU and the NIC can be a major bottleneck. These techniques attempt to alleviate the bottleneck by bringing the CPU and the NIC closer together, or by reducing the overall bandwidth requirement on the channel.

8.2.1 Network Interface Controller Data Caching

A few researchers have proposed putting a modestly sized cache onto the network interface controller in the form of a multi-megabyte DRAM module. Kim, Pai, and Rixner [47] propose to use this cache to store commonly used data pages that are transmitted by a web server. This cache is managed by the OS driver code via two operations: one that tells the NIC to copy a chunk of data to the cache and one that tells the NIC to append a data segment from that cache instead of from main memory. They show that this optimization can improve the performance of a web server by up to 31% due to a 57% reduction in PCI traffic. If the NIC were integrated on the CPU die (as with SINIC; see Chapter 6), it would have a high bandwidth channel to main memory, some or all of which would be available as a cache if the same kernel modifications were made. This approach also eliminates the special NIC operations required for the NIC to manage its cache. Furthermore, the lack of a separate physical cache avoids any coherence problems between kernel buffers and the NIC cache. Finally, the addition of more interfaces to share the traffic load would not reduce the caching benefit as it does in their work because in the on-chip case, each interface would not need its own separate cache.

Yocum and Chase [86] use a NIC cache to enhance packet forwarding performance by up to 60%. Instead of transferring complete packets to memory, only the headers are transferred and inspected by the host CPU. The host can then do analysis on the header and subsequently determine what to do with the payload—i.e. whether to forward the packet on another connection or consume the packet locally. This optimization can be implemented easily by V-SINIC (Chapter 7). The data can simply be buffered in the input FIFO until its destination is determined by the packet forwarding logic, at which time it can be copied to the host, or forwarded to another interface.

8.2.2 Communication Streaming Architecture

Intel recognized that the PCI bus can be a limiting factor in networking performance. In response to the problem, they created a dedicated bus for the network interface which they call Communication Streaming Architecture (CSA). In a traditional Intel-based system, the communication between the CPU and the NIC goes from the CPU via a proprietary link to the memory controller (a.k.a. “north bridge”), then via a second proprietary interface to the PCI bus bridge (a.k.a. “south bridge”) and finally via PCI-X to the network interface. The CSA architecture implements a proprietary bus between the NIC and the memory controller. This optimization is the NIC analogue to the accelerated graphics port (AGP) optimization provided for high performance graphics adapters. While CSA improves the latency and bandwidth to the NIC, it does not address the fundamental growing memory performance gap. The off-chip interface can continue to increase in speed, but this speed does not scale with the speed of the CPU as would be possible if the NIC were on die talking directly to the L2 cache. The advent of PCI Express [6] has obsoleted both the AGP and CSA interfaces, but a revival of this idea may be seen if future generations of PCI Express lack the bandwidth required for future generations of Ethernet.

8.2.3 Direct Cache Access

Intel researchers have proposed a technique called “direct cache access” (DCA) [38] for pushing NIC data into an on-chip cache from an off-chip NIC. DCA modifies the cache coherence protocol to allow an external I/O device to push data into the CPU’s cache. Cache placement of NIC data is natural when the NIC is on the same chip, as in SINIC, and I also find substantial benefits from this optimization in Chapter 5.

The Intel evaluation focuses on prototyping more than simulation, allowing them to look at larger workloads but a more constrained set of hardware configurations.

8.2.4 Integrated and Tightly-Coupled Network Interfaces

Closer integration of network interfaces (NIs)[5] with CPUs has been a prominent theme of work in the area of fine-grain massively parallel processors (MPPs). I borrow from the heritage of earlier these message passing systems. Dally, et al. [20] introduced the J-Machine, a multi-processor computer that uses an on-chip network interface to communicate between nodes. This interface is directly connected to the processor and shares a small memory with the processor. Henry and Joerg [35] investigated a range of placement options, including on- and off-chip memory-mapped NIs. Their most tightly coupled NI is mapped into processor's register file and sends and receives messages using register accesses. Other machines with on-chip network interfaces include the M-Machine [29], *T [62] and IBM's BlueGene/L [64]. Machines with tightly coupled network interfaces include SHRIMP [10] and CM-5 [51].

Mukherjee and Hill [58] also argue for tighter coupling between CPUs and network interfaces for storage-area and cluster networks. They propose several improvements to the network interface that treat the NI more as a core component than a generic peripheral. Specifically, they focus on placing the NI in the coherent memory domain but not on physical integration with the CPU. Notable points that they suggest are that NI device registers be virtually mapped into user-level programs, that network interfaces be moved from the I/O bus to the memory bus, and that NI device registers be cached by the CPU. Accessing device registers via virtual addresses retains protected access to the device without using the operating system kernel because of the protection mechanism in the virtual memory system. Their desire to move the NI from an I/O bus to the memory bus would serve multiple purposes. It would first allow the NI to communicate with the CPU at a much lower latency and a higher bandwidth. In addition, by being attached to the memory bus, the NI can participate in the cache coherence protocol. By participating in the coherence protocol, the NI can now allow the CPU to cache NI device registers, and the NI can cache CPU data. By having cached device registers, the CPU is also able to speculatively access those registers. Finally, cached registers make CPU polling situations much lower cost. Instead of sending out a bus transaction every time the CPU wants to determine if

[5]Network interface controller or NIC is the traditional term used for interfaces that connect local area networks such Ethernet. Network Interface or NI is the conventional term used when referring to the connection to a massively parallel processor system's network.

the NI has data, the transaction will only occur if the NI has changed state. Many of these optimizations could be used with the integrated NIC discussed in Chapter 6 and Chapter 7.

In all of these cases, the primary goal is low user-to-user latency using lightweight protocols and hardware protection mechanisms. In contrast, TCP/IP processing has much higher overhead and practically requires kernel involvement to maintain interprocess protection. This thesis continues in the spirit of this research, but focuses on optimizing the NIC interface for what the kernel would like to see for TCP/IP processing. Now that TCP/IP over off-the-shelf 10GigE can provide bandwidth and latency competitive with, and often better than, special-purpose high-speed interconnects [28], a single CPU device with integrated 10GigE support could serve as both a datacenter server and a node in a high-performance clustered MPP.

On the commercial front, integrated NIC/CPU chips targeted at the embedded network appliance market are available (e.g., [12]); 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 an upcoming version of Sun's Niagara [48] processor, a multithreaded chip multiprocessor designed for network workloads and scheduled to ship in 2005, will have multiple integrated 10 Gbps Ethernet interfaces [22].

8.2.5 TCP Onloading

Most or all of the benefits of TCP offload can also be achieved by dedicating a host processor to protocol processing. This is the theme of an Intel research project known as ETA [72] or "TCP Onloading" [71]. Dedicating a processor to protocol processing, rather than pushing intelligence out to an off-chip NIC, enjoys the performance increases seen by the general-purpose CPU market without effort from the NIC vendor. The end result is that other CPUs are able to communicate with the dedicated CPU at a lower interaction cost than communicating to the NIC directly. I thus consider it an interface optimization, though unlike the other work in this section, it is a software rather than a hardware technique. There is a similarity between TCP onloading and TOE since the protocol processing is separated from normal CPU operation, but because this is strictly a kernel optimization, the protocol processing is still done by the kernel instead of firmware, and major changes to the protocol stack are unnecessary. An integrated NIC argues even more strongly for software innovation within the scope of a homogeneous CMP/CMT general-purpose platform rather than dedicating specialized compute resources to the NIC. Since the TCP onloading work is focused on system software architecture, it is highly complementary to the investigation of system hardware organizations found in Chapter 6 and Chapter 7.

8.3 Copy Avoidance

The key per-byte costs of TCP networking are checksum calculation and copying. This section discusses non-offload techniques that reduce these overheads. The primary focus of many software optimizations is that of zero-copy (more accurately single-copy) receives. It has been implemented in many contexts [15], and continues to be a common goal. In Chapter 7 I describe a zero-copy receive implementation for V-SINIC.

8.3.1 Page Remapping

A well known technique for avoiding copies is known as page remapping (a.k.a. page flipping) [17], where the operating system uses virtual memory techniques to avoid copy operations. On transmit, this is accomplished by making a copy-on-write copy of the page. As long as the user does not write to the page before the kernel is done with it, no copy will take place. For a receive, the kernel simply swaps the page passed in by the user with the page full of data from the kernel. This technique is used in Trapeze [16] to provide zero-copy on standard TCP. The major drawback of this technique is the requirement of page-size MTUs (generally 4kB or 8kB), making it largely unusable on the internet which has a de facto standard MTU of 1500 bytes. Additionally, the required page-table manipulations, while faster than an actual page copy, are not quick in an absolute sense, especially on multiprocessor systems. Page remapping in Trapeze is made more difficult by the fact that the header needs to be placed in a separate page of memory since it is not passed to the application. The difficulty with this lies in the fact that the NIC must have enough intelligence to determine the boundary between the headers and the payload. While this separation is very similar to what I do for the zero-copy mechanism in V-SINIC, but V-SINIC allows this to be done in software and does not require the extra intelligence to exist in the NIC.

8.3.2 Enhanced NIC Buffering

My V-SINIC approach is most closely related to that of Afterburner [21], an experimental NIC that combined significant on-board buffering with a modified protocol stack such that copying packet payloads off of the NIC could be deferred until the destination user buffer was known. The amount of buffer space required is proportional to the product of the network bandwidth and the CPU/NIC latency, and quickly becomes significant. Although Afterburner was relatively closely coupled to the CPU (plugging into a graphics card slot on an HP workstation), it had 1 MB of on-board buffering for a 1 Gbps network. The extremely low latency afforded by on-chip integration allows SINIC to support a similar technique on a 10 Gbps network with substantially less buffering.[6]

8.3.3 Improved Programming Interfaces

IO-Lite [65] and its predecessor `fbufs` [23] provide a new user API that does not have the copy semantics of the traditional sockets interface. The goal of these new APIs is to remove the copies that traditional interfaces require. The system focuses on the fact that data buffers are immutable; they are allocated with initial data, and thereafter, the contents of those buffers cannot be modified. In effect, once data becomes shared, it becomes read-only. To pass data from a sender and a receiver, IO-lite uses a collection of pointer, length pairs (a buffer-aggregate in IO-Lite terms) to represent the data being moved. These aggregates are mutable and have copy semantics. Reads and writes are now done using these mutable buffer aggregates, and the copy is eliminated. The IO-Lite technique is very similar to the kernel's usage of `mbufs`, but goes a step further to provide this interface to the user.

8.3.4 User-Level Network Access

Since traversing the kernel to handle the network protocol stack can incur a significant amount of overhead [30], there have been several proposals for bypassing the kernel and allowing user-space programs to talk directly to the NIC [10, 84, 58, 25].

Level 5 Networks [83] has produced an interesting NIC that is similar to a NIC designed for VIA [25], but instead of specifying a new protocol on top of their NIC, they implement TCP/IP at the user level. This allows them to bypass the kernel

[6]My simulated implementation has up to 380 KB of space in the receive FIFO—256 VNICs times 1514 bytes per packet—but I believe that other performance will not suffer with a smaller FIFO. I intend to experiment with this parameter in the future.

to achieve lower latency communication while still supporting the TCP/IP sockets interface. It also has the added benefit that their accelerator is only needed on one end of a connection. These interfaces have similarities to V-SINIC in that they virtualize the NIC, though they use virtualization differently. They provide per-connection virtualization to provide security and to allow the hardware to demultiplex packets to connections bound to different user processes. V-SINIC on the other hand uses per-packet virtualization to provide parallel access to the packet processing engine which allows deferred copy of the data. This deferred copy allows the demultiplexing to occur in software while still achieving the benefit zero-copy to the destination connection.

8.3.5 Remote Direct Memory Access (RDMA)

Zero-copy behavior can also be achieved using Remote Direct Memory Access (RDMA) [70], a protocol that specifies a mechanism for a user process on one host to refer to memory buffers of a user process on another host. The benefit of this approach is that the users of RDMA pre-register their buffers before the transactions take place so that the kernel can read and write data directly to/from those buffers, removing the need for any data copying. Unfortunately, RDMA requires applications to be rewritten and TOE-like hardware to provide zero-copy efficiency. V-SINIC can provide the same zero-copy benefits to legacy applications without changing protocols and without a complex NIC. If RDMA is still desired, SINIC could provide zero-copy support for it in software without requiring a TOE.

The protocol specifies addresses using offsets within pre-registered buffers, making it relatively straightforward for an implementer to offload this functionality to a sophisticated network interface. The remote NIC has a mapping of the buffer addresses to physical addresses on the remote host, and when data is read or written from the remote host, it can simply DMA to and from those regions of memory. Since this offloads the protocol handling to the NIC, the protocol is very inflexible, and like TOEs, it becomes difficult to do things like tunneling, and filtering. In addition to requiring sophisticated NIC support, RDMA is a significant change to both applications and protocols, and requires support on both ends of a connection.

In addition to the standard RDMA protocol mentioned above, SHRIMP [10, 9] also provided the ability for user processes on one machine to map the memory of user processes on another machine using its user level direct memory access (UDMA).

VIA [25] also supports RDMA for communication between user level processes, though this is built on top of its virtual interfaces and is an optional feature.

8.3.6 Combined Checksum/Copy

In addition to copy, checksum is a costly data-touching operation. It does not receive as much attention though because it is not universal among protocols and is generally easy to deal with. In addition to the checksum offload technique mentioned above, another technique is to combine the checksum and copy loops into a single loop so that the data is only read once [19]. In fact, if the copy must be performed, this technique is effective enough to make checksum offload almost pointless.

8.4 Reducing The Cost Of Interrupts

While per-byte costs of networking can dominate the total overhead of a TCP/IP communications channel for large packet sizes, for small packets, which are more common, it is often the per-packet overhead that dominates [44]. The interruption of the CPU to vector into the networking stack can be a large fraction of the total overhead.

A 10 GigE adapter receiving 1500 byte packets at maximum line rate would receive about 830,000 packets per second. If each packet generated an interrupt, it would be difficult for a CPU to do much else other than service those interrupts. Under such high interrupt load, a CPU would spend a great deal of time dealing with the overhead of servicing of interrupts. Instead of the interrupting the CPU for each one of these packets, one could sacrifice a bit of timeliness for performance by periodically polling the NIC. The polling could happen during a clock timer interrupt (100-1000/s on many machines), for example [80]. Under high load, there should be data at every clock interrupt, and the interrupt overhead is greatly reduced. Linux has a new driver API known as NAPI that supports polling.[7]

Aron and Druschel [4] propose a novel technique to improve timer resolution called "soft timers". While their main focus is to enable rate-based clocking of TCP, their technique is applicable to polling schemes as well. Soft timers provides a software based timer system with a guarantee on minimum timer resolution, but can often provide a much higher resolution. The minimum resolution is achieved by scheduling

[7]NAPI is described in the Linux kernel documentation and can be found in the Linux source tree under `Documentation/networking/NAPI_HOWTO.txt`.

a timer interrupt to occur at the minimum clock resolution. Improved resolution is achieved by also checking the timers on other transitions into the operating system, including other interrupts and system calls. These other events happen frequently on network server type machines, and while these disruptions might normally be thought of as overhead, their existence can be taken advantage of to increase the timer resolution. Without this technique, it would be necessary to increase the rate of timer interrupts to improve resolution.

The problem with polling is that when the NIC is idle, the CPU is wasting quite a bit of time doing long-latency device register accesses to the NIC to find out that there is nothing to do. If the CPU were busy, this could impose a significant penalty on the CPU performance. One might try to use an interrupt mechanism at low load and switch over to a polling mechanism at high load, but determining when to switch over can be relatively complex. Many modern adapters instead use a technique referred to as interrupt coalescing or moderation. In this technique, the NIC doesn't interrupt on every packet reception, but delays the interrupt for each packet by a specified amount of time in hopes that there will be other interrupts during the delay. Another way to apply this technique is to not interrupt on a per-packet basis, but instead interrupt when a predetermined fraction of the receive buffer is filled. Intel's Pro/1000 series of Gigabit Ethernet adapters employ this interrupt moderation technique [43].

Chapter 9

Future Work

There are a number of directions in which this dissertation could be extended, both in expanding the analysis of this work and in exploring new ideas that build upon it. Specific areas include: better benchmarking; extending SINIC and V-SINIC; investigating the implications of placing network data directly into the cache; looking at alternatives to interrupts and polling for notification of packet arrival; optimizations for network packet processing; software architecture changes to take advantage of new NICs; and universal messaging support.

9.1 Benchmarks and Analysis

One element of future work is expanding the benchmark suite to include additional macro-benchmarks. File servers, virtual private network (VPN) gateways, routers, firewalls, proxies, and distributed denial of service (DDOS) detectors are among the many applications that could benefit from a general purpose processor with an integrated NIC like SINIC or V-SINIC. As an example, firewalls are very commonly implemented using a high-performance server containing a few high-performance NICs. With the DMA-descriptor architecture of these NICs, it is necessary to copy all data into system memory, even for those packets that you choose to filter. V-SINIC on the other hand can be used to allow the firewall to copy just the headers, examine them, and either discard the payload or copy the payload, depending on the results of the filter. A further extension to V-SINIC could allow the payload to be moved directly from the input FIFO to the output FIFO to save the CPU the overhead of moving the data itself (similar to the payload caching of Yocum and Chase [86]).

In addition to having more benchmarks, a quantitative comparison of the performance of an integrated NIC with a TOE is highly desirable. While SINIC/V-SINIC is the primary focus of this research, software architectures like TCP onloading [71] should certainly be investigated. The software architecture of a TOE is very similar to onloading, albeit with extra overhead between the CPU and the TOE for DMA. This similarity of the software architectures coupled with the flexibility of the M5 simulation environment ought to allow me to emulate a TOE by changing the communication parameters between the application CPU and the dedicated CPU, thereby making the dedicated CPU TOE-like.

9.2 Future SINIC/V-SINIC Work

With V-SINIC, I would like to performance tune my zero-copy implementation and further explores the SINIC/V-SINIC design space, including sensitivity analysis of the SINIC access latency and number of VNICs available for V-SINIC. Open issues include how best to support encryption and decryption of network traffic and how to virtualize SINIC for virtual-machine systems.

9.3 Cache Interaction

I have begun to investigate the potential for NIC-based header splitting to selectively DMA only packet headers into the on-chip cache (see Chapter 5). The "header splitting" technique employed there 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. There is clearly room for more intelligent policies that can get both the full benefit of payload cache placement when possible and avoid its pollution effects in other situations. These policies might 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 [87].

My research group has investigated an alternative cache architecture known as the indirect-index cache (IIC) [34]. This cache architecture approximates a fully associative cache by sequentially looking up cache tags and data. The tag is first looked up in a normal associative structure; if it is not found there, is looked up in a chained hash-table-like structure. On a cache hit, a pointer to a location in the data array is produced. This pointer can point to any location in the data array, and is allocated separately from the tags. The NIC FIFO packet data reside in the data array, with the FIFO represented by an array of tag-like pointers into the data array. If properly aligned, the copy of data from the NIC FIFO to the CPU could actually be a simple transfer of ownership of the block from the NIC FIFO to the cache tag array. This could reduce the number of copies occurring to one less than what is traditionally called a “zero-copy” implementation!

9.4 Event Notification

Another opportunity lies in the interaction of packet processing and CPU scheduling. The cost of interrupts on protocol traffic is well known [80]. Polling and Interrupt coalescing are two well known techniques for mitigating the cost of these interrupts (see Chapter 8). Unfortunately, latency-sensitive applications suffer from these sorts of optimizations. An on-chip NIC co-designed with a multi-threaded CPU could possibly leverage a hardware thread scheduler to provide low-overhead notification. This would be accomplished by having the CPU automatically spawn threads as each packet arrived much like in earlier MPP machines [20, 62]. q In order to provide the many threads that would be required for the thread per packet architecture described above, the virtual context architecture (VCA), developed by my colleagues and I [63], could be employed. VCA can support potentially dozens of active hardware threads, and when coupled with a hardware thread scheduler, could be extended to emulate having a virtually infinite number of hardware threads. One could imagine a system with dozens of threads, each for different sorts of events, sleeping while waiting for events from external devices and waking up on demand when an event occurs.

9.5 Packet Processing Optimizations

In the basic SINIC model, the FIFOs are accessed from any of the on-die CPUs through standard PIO requests. If one CPU was dedicated to packet processing, that CPU could be optimized for this task. For example, the read to get result of the block copy is accomplished with an uncached access. Since uncached accesses are generally considered non-speculative, they represent a serialization point in the hardware. There is really no reason for them to have this property since they do not have any side effects. Non-serializing accesses could be achieved either by special-casing the memory-mapped locations used by the SINIC control registers or even mapping the SINIC registers into the CPU’s register file. Additionally, there is no specific reason that the block-copy/checksum unit work with data moving between the FIFOs and memory. It could move data between different locations of memory, or between the FIFOs themselves, or could be used to checksum an arbitrary portion of memory.

The dedicated CPU can be extended to optimize several common receive-side operations. The first—and potentially most useful for TCP/IP—is header matching and prediction. A great deal of TCP/IP processing is spent determining what protocols are involved and to which connection a particular packet belongs. Instead of sequentially processing each portion of each header until the correct connection is determined, partial headers for a number of the most common connections can be maintained in a content-associative memory structure. If a packet matches one of these partial headers, it can be immediately demultiplexed to the proper connection. Depending on how much information is presented in the partial header and on constraints on the size of the matching hardware, an exact match may not be determined; instead, the match can be used as a prediction. An example of the usage of this header matching for TCP/IP would be to build a partial headers with the proper protocol types, source IP address, destination IP address, source port, and destination port. When a packet arrives, it could be matched against many of the most common headers to find a match. The information in these partial headers properly demultiplexes a packet to its correct connection, but does not ensure that the packets are correctly formed, or received in the proper order. A second partial header containing information about the particular connection, such as the next sequence number, proper flags, etc. could be used to explicitly match the incoming packet. If this second comparison matches, the common case of receiving an in-order, well-formed packet can be quickly verified, whereas the uncommon out-of-order, or exception case can be checked using the same sequential method in use today. The hardware involved to implement this header matching would simply be a set of offset, length, value tuples that would be searched for the demux case and matched using a wide comparator for the verification case.

In general, it can be difficult to schedule I/O across multiple CPUs. The dedicated CPU could do the demultiplexing of the packets using its header matching functionality, determine the best CPU for processing the given connection, and pass that connection off to that CPU. This would be similar to Microsoft's receive side scaling (RSS) [56], though it would be more flexible and potentially more intelligent.

Other optimizations can be implemented on the transmit side. The first is header template caching, the transmit analogue of header matching and prediction. A partial header can be generated for each connection and stored in a hardware table. Then, in a single operation, the template can be combined with details for a particular packet such as length and sequence number. The resulting completed header can either be placed in memory for further analysis or potentially copied directly to the FIFO. After the header is placed in the FIFO, the payload data can be copied directly to the FIFO using the block copy operation. This sort of optimization would achieve the very similar results to TCP Segmentation Offloading (see Chapter 2) due to the fact that the per-packet cost has been significantly reduced, and the generation of full header information is amortized across the transmission of many packets.

Unfortunately, for protocols like TCP, it is generally necessary to save a copy of the payload data in memory in the case of a transmission problem. In the case of very low latency communications, a software optimization can be performed where the result of the I/O is not communicated to the user program until the data is acknowledged by the receiver. If this optimization is not possible, the block copy operation can be used for this copy. Finally, if the transmit FIFO were large enough, the packet could be buffered there until the acknowledgment is received.

9.6 Universal High-Performance Messaging Support

While this work has targeted the SINIC architecture specifically for TCP/IP over Ethernet, SINIC could also be applied to other packet-based communications, leading to universal interface for messaging. This support could have very wide applicability, as most I/O in modern systems is packet based, including both network I/O and disk I/O. One the primary jobs of NICs and SCSI adapters is to convert from memory buffers to packets. Instead of reimplementing this functionality over and over for every new I/O adapter, SINIC could provide this functionality once on the CPU die in such a way as to benefit both on- and off-chip devices. New interconnects like PCI Express [6] and HyperTransport [40] are packet-based at the lowest level. The traditional access patterns of reading and writing data to memory locations are actually mapped on top of this packet-based architecture, and then packet-based communications like Ethernet and SCSI are mapped on top of that. Removing this extra layer, giving raw access to the low-level packets could dramatically change the way people consider designing peripherals.

The simple FIFO pair model coupled with the ability to easily use any physical media at the peripheral makes it possible for the same host hardware to be used in many different applications. To show the applicability of this approach to scientific parallel computing, the queues of multiple processors could be connected by a reliable crossbar switch fabric, resulting in a simple network with Infiniband-like characteristics. The dedicated CPU on each processor would provide a virtualized queue interface between pairs of processes to allow MPI to be implemented very close to the hardware with minimal additional software support. The result would be a very high-bandwidth, low-latency interconnect for scientific computing.

Chapter 10

Conclusions

Architects can use the research described and the lessons learned in this dissertation to design systems better adapted to the demands of networking. In addition, they can use the tool developed for this research to conduct further architectural studies.

The first study of this dissertation investigated the impact of placing a conventional 10 Gigabit Ethernet NIC at different locations in the memory hierarchy ranging from the traditional location on a PCI peripheral bus to an on-die location. I find that tighter integration has clear benefits in terms of achieved bandwidth and CPU utilization. These benefits are a direct result of the higher bandwidth and lower latency available when compared to conventional NIC attachment. The study revealed that simply integrating a traditional NIC onto the CPU die can lead to performance improvements of more than 2x over conventional peripheral based designs on some benchmarks. Another part of this study investigated the implications of giving the NIC access to the on-chip memory hierarchy. Here, I determined that cache placement of DMA data has the potential for tremendous wins in performance in some cases due to a greatly reduced cache miss rate while not affecting performance though possibly increasing cache miss rate in others.

Since on-chip integration with cache attachment was an overall win in terms of performance, the second study investigated new NIC architectures based on this configuration. The designs for these new configurations were driven by the different characteristics of the communication between the NIC and CPU. The resulting new NIC architecture was SINIC, a design that is optimized for the low-latency interaction provided by integration. SINIC is simpler than conventional NICs in that it removes DMA descriptors and their associated overhead and complexity. Instead of using DMA descriptors, SINIC provides direct access to both of its FIFOs with a block-copy engine. It is the responsibility of the host CPU to initiate the transfer of each buffer between the NIC and the CPU. Despite the relative simplicity of SINIC, detailed full-system simulation results show that SINIC performs as well as or better than a conventional NIC given the same level of integration. In addition to the basic SINIC architecture, I also presented a novel approach to extending SINIC's FIFO-based interface to allow packet-level parallelism both on transmit and receive. By associating a set of "virtual" FIFO registers to each packet, the V-SINIC interface allows lockless concurrent packet transmission on multiprocessors and enhanced parallelism in receive packet processing. V-SINIC also enables a deferred-copy technique that supports a straightforward implementation of zero-copy receive handling, which I have implemented in the Linux 2.6 kernel. With this implementation, I did observe a reduction in the overall time spent copying data. Unfortunately, do to what I believe are deficiencies in my software architecture, this did not result in an improvement of overall performance.

In addition to the studies performed, the work provides researchers with a tool to explore entirely new system architectures for networked hosts—architectures unconstrained by traditional designs. The sorts of studies conducted in this dissertation are typically difficult to evaluate because the most radical architectures cannot be studied using hardware prototyping or emulation. They must be either implemented, an extremely costly proposition, or simulated, a more practical one. Since there was no tool available that allowed me to conduct the research in this dissertation, I developed the M5 simulator [7], an environment specifically targeting networked systems. The development of this system represents a significant amount of work, several man-years, and it is, as a result, unique in its ability. No other simulator, academic or commercial, supports a detailed network I/O model capable of providing realistic timing measurements. To provide the deterministic, repeatable results in this dissertation, M5 is capable of simulating server and client systems along with a simple network in a single process. M5 uses full-system simulation to capture the execution of both application and OS code and models the hardware platform with enough fidelity that it can boot a variety of operating systems, including Linux and FreeBSD, without modification. This environment is publicly available [52] to promote further research in this area.

APPENDICES

Appendix A

Simple Integrated Network Interface Controller Datasheet

This datasheet describes the simple integrated network interface controller (Chapter 6) and its virtualization extension (Chapter 7). The datasheet presents the technical data necessary to re-implement either the device model or the device driver. It would be particularly useful for porting the device driver to another operating system.

Both SINIC and V-SINIC do not exist in the real world, their current implementation exists only within the M5 simulator. This has implications on how the device is configured and how statistics are collected.

SINIC and V-SINIC are pre-configured by the simulator before the operating system boots. With a real device, configuration registers would be set by the operating system during the device driver configuration to select the features that the operating system desires to use. Because it is far easier to modify the simulator configuration than the operating system configuration, this process is reversed. The simulator will set the configuration options desired by the device, and the driver will read those values to determine how it should function.

Normally, a NIC would report statistics to the operating system so that information about the performance of the network would be available, but since SINIC and V-SINIC have no real-world analogue and are intended to only be run in simulation, we rely on the simulator itself to gather statistics directly from the device model itself. This greatly simplifies the driver implementation, and provides far more accurate statistics about how the device is actually performing. If at some point, it became desirable to have the operating system tune itself to some sort of NIC statistic, then it would become necessary to expose those statistics in the device's interface.

The rest of this appendix describes the functional operation of SINIC/V-SINIC and the programming interface presented to the OS.

A.1 Functional Description

SINIC and V-SINIC can operate on either virtual[1] or physical addresses, though it is limited to accessing 1TB of either address space. Because it is designed for kernel-based TCP/IP processing, it does not face the address translation and protection issues of user-level network interfaces.

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

The RxDone and TxDone registers provide status information on their respective FIFOs. Each register indicates the number of packets in the FIFO, whether the associated copy engine is busy, whether the last copy operation initiated on the FIFO completed successfully, and the actual number of bytes copied.[2] TxDone also indicates whether the transmit FIFO is full. RxDone includes several additional bits. One bit indicates whether there is more data from the current packet in the FIFO. Another set of bits indicates whether the incoming packet is an IP, UDP, or TCP packet, and whether SINIC's calculated checksum matched the received packet checksum.

SINIC implements a single copy engine per FIFO, though these engines share a single cache port, so only one can perform a transfer in any given cycle. As a result, the CPU must wait for the previous copy to complete before initiating another copy. Because individual buffer transfers are relatively fast, the driver simply busy-waits when it needs to perform multiple copies. SINIC enables more efficient synchroniza-

[1]Virtual addresses are available to SINIC because it on chip and has access to the on-chip TLB. Currently, the SINIC model is not properly tied into the processor TLB, so using SINIC in this mode should be considered less accurate from a timing perspective. There are also implications with TLB shootdown that one must consider if properly implementing virtual address mode.

[2]This last value is useful as it allows the driver to provide the allocated buffer size as the copy length to the receive FIFO and rely on SINIC to copy out only a single packet even if the packet is shorter than the buffer.

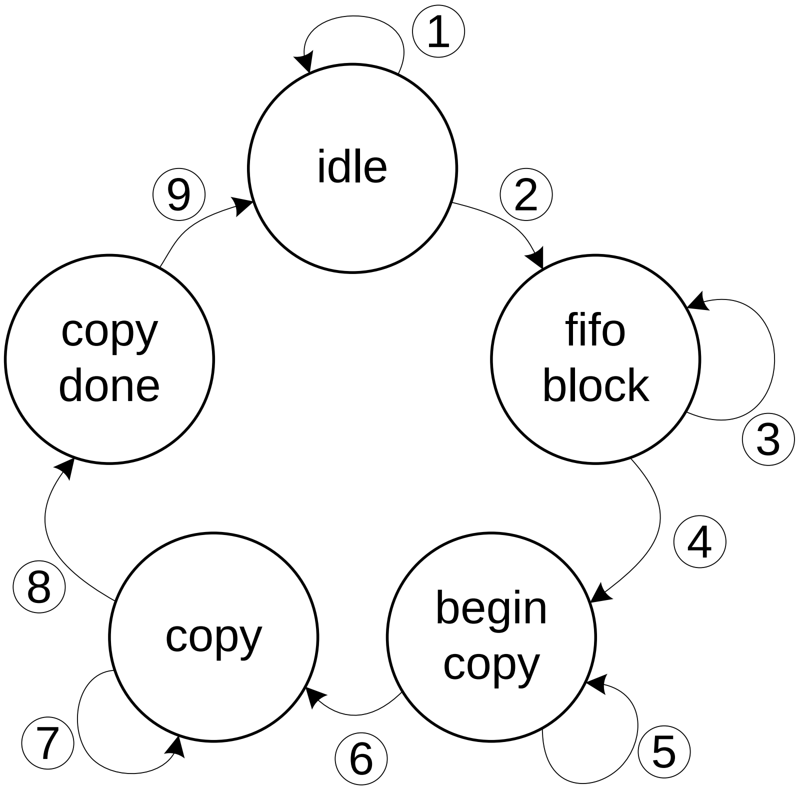


Figure A.1: SINIC State Machine - State machine implemented by the SINIC device model for both receive and transmit. (1) No DMA pending, waiting for RxData/TxData. (2) RxData/TxData written, requesting DMA. (3) Rx FIFO empty/Tx FIFO full. (4) Data available in Rx FIFO / Tx FIFO has room for new data. (5) DMA Interface not ready. (6) DMA Interface ready. (7) Copy busy. (8) Copy complete. (9) RxDone/TxDone updated.
This state machine is intended to represent the various logical states that the device goes through. In some instances multiple states may be processed at one time.

tion through two additional status registers, RxWait and TxWait. These registers return the same status information as RxDone and TxDone, respectively, but a load to either of these registers is not satisfied by SINIC until the corresponding copy engine is free. Thus a single load to RxWait replaces a busy-wait loop of loads to RxDone, reducing memory bandwidth and power consumption.

A.1.1 Receive State Machine

With SINIC, as opposed to a conventional NIC, there are no descriptors. In order to receive a packet, the posts a buffer with a PIO write to RxData. It is possible, and desirable, to post this buffer long before the packet arrives to allow SINIC to overlap the copy operation with computation on the CPU. When the CPU is ready to process a packet, it simply calls RxDone to determine the status of the current packet. It is generally not necessary to busy wait on RxDone or RxWait, though if a hardware thread were dedicated to receive, this would be the condition to busy wait on. If the More bit is set on a packet, this indicates that there is more data in the packet and that the next call to TxData will receive that data. With SINIC, packets are processed fully in-order (unlike V-SINIC, as described below).

A.1.2 Transmit State Machine

Transmit operates in a similar fashion to receive. When a packet must be transmitted, the device driver simply loops over each portion of the packet buffer initiating a transmit with a PIO write to TxData, setting the More bit if there is more data for the packet. With the More bit set, the next write to TxData will correspond to the next portion of the packet. As with receive, SINIC will only process transmitted packets in-order, forcing a complete packet to be written before the next one can begin. Before the next portion of the packet, or another packet can be sent with TxData, the driver must busy-wait on either TxDone or TxWait to ensure proper completion of the previous TxData operation. It is possible to make subsequent calls to TxData, but if one is called before the previous one completes, the results are undefined. It is not necessary for the wait to occur immediately, the wait can occur just before the next packet is ready to transmit, allowing the wait to overlap other instructions.

A.2 V-SINIC Operation

As long as each packet is copied to (or from) memory in its entirety before the next packet is processed, a single blocking copy engine per FIFO is adequate. However, there are situations—such as the zero-copy optimization described in Section 7.2—where it is useful to begin processing a packet before the preceding packet is completely copied into or out of the FIFO. This feature is particularly desirable for chip multiprocessor systems, where packet processing can be distributed across multiple CPUs.

The virtualized SINIC mode, known as V-SINIC, enables packet-level parallelism by providing multiple sets of RxData, RxDone, TxData, and TxDone registers for each FIFO and dynamically associating different register sets with different packets.

This provides each in-process packet with its own virtual interface, referred to as a virtual NIC or VNIC. V-SINIC still has only one copy engine per direction, but each engine is multiplexed dynamically among the active VNICs. V-SINIC supports one outstanding copy per VNIC; once a copy is initiated on a VNIC, that VNIC will be marked busy until it acquires the copy engine and completes the copy.

Although the V-SINIC extensions to both the receive and transmit FIFOs are conceptually similar, they differ slightly in details and significantly in usage. See below for details.

A.2.1 Receive State Machine

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

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

A.2.2 Transmit State Machine

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

A.3 Register Set

The following set of registers is used for both SINIC and V-SINIC, though for V-SINIC, the registers are replicated multiple times, once for each VNIC.

off	tag	width	virtual	access	usage
00h	Config	32b	no	RW	Configuration Register
04h	Command	32b	no	WO	Command Register
08h	IntrStatus	32b	no	R/W	Interrupt Status
0Ch	IntrMask	32b	no	RW	Interrupt Mask
10h	RxMaxCopy	32b	no	RO	Maximum bytes copied by RxData
14h	TxMaxCopy	32b	no	RO	Maximum bytes copied by RxData
18h	RxMaxIntr	32b	no	RO	Maximum number of receives processed per interrupt
1Ch	—	32b			Reserved
20h	RxFifoSize	32b	no	RO	Receive FIFO capacity in bytes
24h	TxFifoSize	32b	no	RO	Transmit FIFO capacity in bytes
28h	RxFifoMark	32b	no	RO	Receive FIFO high watermark
2Ch	TxFifoMark	32b	no	RO	Transmit FIFO low watermark
30h	RxData	64b	yes	RW	Initiate receive DMA
38h	RxDone	64b	yes	RO	Receive DMA status (poll)
40h	RxWait	64b	yes	RO	Receive DMA status (block)
48h	TxData	64b	yes	RW	Initiate transmit DMA
50h	TxDone	64b	yes	RO	Transmit DMA status (poll)
58h	TxWait	64b	yes	RO	Transmit DMA status (block)
60h	HwAddr	64b	no	RO	MAC Address

68-FFh	—		—		Reserved

A.3.1 Config Register

Tag:	Config	Size:	32 bits	Reset:	M5 Config
Offset:	00h	Access:	Read Only		

Configuration variables set up by the simulator device model and passed to the device driver code. Most are actually configurable by the runtime configuration on a per device basis (See Section A.4). They are intended to inform the device driver of how the simulator user would like the driver to behave.

bit	tag	usage
31-14	—	Reserved (read zero)
13	ZeroCopy	Enable zero copy. Tell the V-SINIC device driver to use the zero copy optimization. Only valid if the Vnic bit is set. Cannot be used in conjunction with DelayCopy.
12	DelayCopy	Enable delayed copy. Tell the V-SINIC device driver to use the delay copy optimization. Only valid if the Vnic bit is set. Cannot be used in conjunction with ZeroCopy.
11	RSS	Enable receive side scaling. If the RxThread is active, tell the device driver to use some sort of intelligent receive side scaling. The implementation is driver dependent.
10	Vnic	Enable V-SINIC. If this bit is set, then V-SINIC is enabled, and this entire register set is replicated once for each VNIC.
9	RxThread	Enable receive threads. Tells the device driver to use multiple receive threads to allow packet reception on multiple processors.
8	TxThread	Enable transmit thread. Tells the device driver to use multiple transmit threads to provide lockless access to the device driver if V-SINIC is not enabled.[3]
7	Filter	Enable receive filter.[4]
6	Vlan	Enable vlan tagging. [5]
5	Vaddr	Enable virtual addressing. Tells the device driver that it is OK for virtual addresses to be sent to the device.

[3]This option is not a good idea if V-SINIC is enabled because the device itself can handle multiple simultaneous transmits allowing the driver to be lockless anyway.

[4]Currently not implemented. Causes the simulator to call `panic()`.

[5]Currently not implemented. Will cause the driver to `panic()`.

4	Desc	Enable transmit/receive descriptors.[6]
3	Poll	Enable polling.[7]
2	IntEn	Enable interrupts.
1	TxEn	Enable transmit.
0	RxEn	Enable receive.

A.3.2 Command Register

Tag:	Command	Size:	32 bits	Reset:	N/A
Offset:	04h	Access:	Write Only		

Rather than requiring a load, mask, and store of the register, this register is write only. A value of one in any field will trigger the desired operation. A value of zero will be ignored.

bit	**tag**	**usage**
31-2	—	Reserved (must be zero)
1	Intr	Cause the device to send a interrupt.
0	Reset	Tell the device to reset.

A.3.3 Interrupt Status Register

Tag:	IntrStatus	Size:	32 bits	Reset:	00000000h
Offset:	08h	Access:	Read/Write		

An interrupt is signaled if the condition listed below happens and its corresponding IntrMask bit is set. The device model may chose to implement some sort of interrupt moderation, so an interrupt condition is not guaranteed to trigger an interrupt.

A read of the interrupt status register will report the interrupt status of the device. A read will cause all status bits to be cleared. A write will cause those status bits that are set to 1 to be cleared.

bit	**tag**	**usage**

[6]Currently not used. Intended for a future version of the device model that supports descriptors. This would be useful for comparison purposes.

[7]Tells the device driver to use polling instead of interrupts. Has no effect on the device/

63-9	—	Reserved (must be zero)
8	Soft	Software interrupt. The software interrupt command was called.
7	TxLow	The TX FIFO dropped below watermark. This interrupt condition is only triggered the first time the TX FIFO passes below the watermark following a TxFull.
6	TxFull	The TX FIFO is full.
5	TxDMA	A TX DMA completed. May be coalesced.
4	TxPacket	A packet transmitted on the wire. May be coalesced.
3	RxHigh	The RX FIFO went above the high watermark. This interrupt condition is only triggered the first time the RX FIFO rises above the watermark following an RxEmpty.
2	RxEmpty	The RX FIFO is empty.
1	RxDMA	An RX DMA completed. May be coalesced.
0	RxPacket	A packet was received. May be coalesced.

A.3.4 Interrupt Mask Register

Tag:	IntrMask	Size:	32 bits	Reset:	M5 Config
Offset:	0Ch	Access:	Read/Write		

The initial value of this register depends on the configuration of the simulator. A read of this register will return the current interrupt mask. A write of this register will set the current interrupt mask. While the device may set an interrupt condition, the interrupt will only be signaled if the interrupt mask bit for that interrupt is set. The bits of this register correspond directly to the IntrStatus bits.

A.3.5 Receive Maximum Copy Register

Tag:	RxMaxCopy	Size:	32 bits	Reset:	M5 Config
Offset:	10h	Access:	Read Only		

Reports the maximum number of bytes that will be copied per RxData operation. This value is set by the device model and is currently configurable within the simulator. See Section A.4.

A.3.6 Transmit Maximum Copy Register

Tag:	TxMaxCopy	Size:	32 bits	Reset:	M5 Config
Offset:	14h	Access:	Read Only		

Reports the maximum number of bytes that will be copied per TxData operation. This value is set by the device model and is currently configurable within the simulator. See Section A.4.

A.3.7 Maximum Receives per Interrupt Register

Tag:	RxMaxIntr	Size:	32 bits	Reset:	M5 Config
Offset:	18h	Access:	Read Only		

Indicates the number of packets that should be received per interrupt of the device driver. This is only used by the device driver and is intended to prevent the device driver from spending too much within an interrupt context, thereby preventing other code from running. See Section A.4.

A.3.8 Receive FIFO Size Register

Tag:	RxFifoSize	Size:	32 bits	Reset:	M5 Config
Offset:	20h	Access:	Read Only		

Reports the size of the receive FIFO to the device driver. The size of the receive FIFO is simulator run time configurable on a per device basis. See Section A.4.

A.3.9 Transmit FIFO Size Register

Tag:	TxFifoSize	Size:	32 bits	Reset:	M5 Config
Offset:	24h	Access:	Read Only		

Reports the size of the transmit FIFO to the device driver. The size of the transmit FIFO is simulator run time configurable on a per device basis. See Section A.4.

A.3.10 Receive FIFO High Watermark

Tag:	RxFifoMark	Size:	32 bits	Reset:	M5 Config
Offset:	28h	Access:	Read Only		

Reports the device receive FIFO high watermark value. It is used to determine when to send the RxMark interrupt to the device (see Section A.3.3). The receive FIFO high watermark value is simulator run time configurable on a per device basis. See Section A.4.

A.3.11 Transmit FIFO Low Watermark

Tag:	TxFifoMark	Size:	32 bits	Reset:	M5 Config
Offset:	2Ch	Access:	Read Only		

Reports the device transmit FIFO low watermark value. It is used to determine when to send the RxMark interrupt to the device (see Section A.3.3). The transmit FIFO high watermark value is simulator run time configurable on a per device basis. See Section A.4.

A.3.12 Initiate Receive DMA

Tag:	RxData	Size:	64 bits	Reset:	0000000000000000h
Offset:	30h	Access:	Read/Write		

The receive data register contains the status of the last issued transmit DMA instruction which was initiated by the RxData register.

bit	**tag**	**usage**
63-61	—	Reserved (must be zero)
60	Vaddr	Addr is virtual and should be translated by the device
59-40	Len	Length or array pointed to by Addr. This is the number of bytes to copy from the current incoming packet. This register allows up to 1MB to be specified, but the maximum value is limited by RxMaxCopy. A copy larger than RxMaxCopy has undefined results (currently the simulator will `panic()`).
39-0	Addr	Address of buffer to receive data. Refers to up to 1TB of address space. This does place limitations on virtual addresses.

A.3.13 Receive DMA Done

Tag:	RxDone	Size:	64 bits	Reset:	0000000000000000h
Offset:	38h	Access:	Read Only		

The receive done register contains the status of the last issued transmit DMA instruction which was initiated by the RxData register.

bit	tag	usage
63-48	—	Reserved
47-32	Packets	Number of unprocessed packets in the receive queue
31	Busy	The receive DMA engine is currently busy with a transaction
30	Complete	The current transaction is complete making the rest of the bits in this register valid
29	More	The current DMA did not receive all of the data available to the current packet, at least one more DMA is required to get the rest of the data for this packet
28-26	—	Reserved
25	TcpError	The current packet has a TCP checksum error
24	UdpError	The current packet has a UDP checksum error
23	IpError	The current packet has an IP checksum error
22	TcpPacket	The current packet is a TCP packet
21	UdpPacket	The current packet is a UDP packet
20	IpPacket	The current packet is an IP packet
19-0	CopyLen	If More is set: CopyLen contains the number of bytes that remain in the current packet and must be copied in future DMAs. (The buffer was filled) If More is not set: CopyLen contains the number of bytes that were copied in the previous DMA Either length can refer to up to RxMaxLen

A.3.14 Receive DMA Wait – RxWait

Tag:	RxWait	Size:	64 bits	Reset:	0000000000000000h
Offset:	40h	Access:	Read Only		

RxWait exports the exact same interface as RxDone and is functionally equivalent. However, RxWait is a blocking operation, and the result will not return until the

Complete bit is set. In order to prevent blocking, RxWait will not wait indefinitely, it will time-out in order to allow the CPU to do other processing in the case of an error or very long latency operation. Therefore, it is necessary to construct a busy loop around TxWait as would be necessary for RxDone.

N.B. RxWait is currently not properly implemented in the M5 simulator and is actually equivalent to RxDone. This is because the M5 simulator CPU model that I used requires the data result before the cache operation completes, not after as would be correct. With the execute-in-execute model, it should be rather straight forward to properly implement RxWait.

A.3.15 Initiate Transmit DMA

Tag:	TxData	Size:	64 bits	Reset:	0000000000000000h
Offset:	48h	Access:	Read/Write		

The transmit data register contains the status of the last issued transmit DMA instruction which was initiated by the TxData register.

bit	tag	usage
63	More	Complete packet not provided. The next TxData will append more data to the current packet. When more bit is not set, the full packet has been transferred to the device and it will be marked for transmit.
62	Checksum	Calculate TCP/UDP/IP checksums for the outgoing packet. This bit is only useful for the final operation on any given packet. (i.e. the More bit must not be set.)
61	—	Reserved (must be zero)
60	Vaddr	Addr is virtual and should be translated by the device.
59-40	Len	Length or array pointed to by Addr. This is number of bytes to copy to the current outgoing packet. This register allows up to 1MB to be specified, but the maximum value is limited by TxMaxCopy. A copy larger than TxMaxCopy has undefined results. (Though, currently the simulator will `panic()`)
39-0	Addr	Address of buffer to transmit data. Refers to up to 1TB of address space. This does place limitations on virtual addresses.

A.3.16 Transmit DMA Done

Tag:	TxDone	Size:	64 bits	Reset:	0000000000000000h
Offset:	50h	Access:	Read Only		

The transmit done register contains the status of the last issued transmit DMA instruction which was initiated by the TxData register.

bit	tag	usage
63-48	—	Reserved
47-32	Packets	Number of untransmitted packets in the Tx FIFO
31	Busy	The transmit DMA engine is currently busy with a transaction. If this bit is set and a write to TxData occurs, the result is undefined (currently results in a simulator `panic()`.
30	Complete	The current transaction is complete making the value of CopyLen valid, and freeing the state machine for another transmit.
29	Full	The Tx FIFO is full, the transmit state machine is stalled. For V-SINIC, unused VNICs can still queue transmits, but none will be processed until the FIFO sends a packet out the interface.
28	Low	The Tx FIFO is below the low watermark. This value is always set if the Tx FIFO is below the watermark, unlike the TxLow interrupt which only signals the first time it drops below the low watermark. This value is not VNIC specific.
27-20	—	Reserved (must be zero)
19-0	CopyLen	Valid only if the current transfer is Complete. Reports the number of bytes copied to the current outgoing packet. Currently, the transmit state machine will always transmit the full amount requested, so this value should always equal the CopyLen value set in TxData

A.3.17 Transmit DMA Wait

Tag:	TxWait	Size:	64 bits	Reset:	0000000000000000h
Offset:	58h	Access:	Read Only		

TxWait exports the exact same interface as TxDone and is functionally equivalent. However, TxWait is a blocking operation, and the result will not return until the Complete bit is set. In order to prevent blocking, TxWait will not wait indefinitely, it will time-out in order to allow the CPU to do other processing in the case of an error or very long latency operation. Therefore, it is necessary to construct a busy loop around TxWait as would be necessary for TxDone.

N.B. TxWait is currently not properly implemented in the M5 simulator and is actually equivalent to TxDone. This is because the M5 simulator CPU model that I used requires the data result before the cache operation completes, not after as would be correct. With the execute-in-execute model, it should be rather straight forward to properly implement TxWait.

A.3.18 Hardware Address

Tag:	HwAddr	Size:	64 bits	Reset:	M5 Config
Offset:	58h	Access:	Read Only		

This register provides the simulator provided value for the Ethernet address of this device.

bit	tag	usage
63-48	—	Reserved
47-40	HwAddr[0]	The first byte of the Ethernet address
39-32	HwAddr[1]	The second byte of the Ethernet address
31-24	HwAddr[2]	The third byte of the Ethernet address
23-16	HwAddr[3]	The fourth byte of the Ethernet address
15-8	HwAddr[4]	The fifth byte of the Ethernet address
7-0	HwAddr[5]	The sixth byte of the Ethernet address

A.4 M5 Simulator Model Configuration Options

These are the options that the M5 simulator provides to the SINIC/V-SINIC device models and are used to configure the various properties of the SINIC/V-SINIC device. These options are configured at runtime using the python configuration system and enable the user to configure multiple devices per system independently. They do not change during the execution of the simulation.

option	type	default	usage

hardware_address	EthernetAddr	Auto[8]	Ethernet Hardware Address
clock	Clock	0ns	State machine processor frequency. Currently not implemented.
dma_read_delay	Latency	0ns	If there is no timing memory hierarchy, use this fixed delay for DMA reads.
dma_read_factor	Latency	0ns	If there is no timing memory hierarchy, this per-byte delay is added to each DMA read.
dma_write_delay	Latency	0ns	If there is no timing memory hierarchy, use this fixed delay for DMA writes.
dma_write_factor	Latency	0ns	If there is no timing memory hierarchy, this per-byte delay is added to each DMA write.
dma_no_allocate	Bool	True	Determines whether an allocation should occur in the cache on read. Only useful for configurations where the device is on the processor side of a cache.
rx_delay	Latency	1μs	Receive Delay. Currently unused.
tx_delay	Latency	1μs	Transmit Delay. Currently unused.
rx_fifo_size	MemorySize	512kB	Set the maximum size of the receive FIFO. Also sets the RxFifoSize device register (see Section A.3.8).
tx_fifo_size	MemorySize	512kB	Set the maximum size of the transmit FIFO. Also sets the TxFifoSize device register (see Section A.3.9).
rx_filter	Bool	True	Enable Receive Filter. Currently has no effect.
intr_delay	Latency	10μs	Interrupt delay used for coalescing delayable interrupts.
rx_thread	Bool	False	Enable dedicated kernel threads for receive. Also sets the RxThread bit of the Config register (see Section A.3.1).

[8]The simulator will automatically provide an Ethernet address to the device

tx_thread	Bool	False	Enable dedicated kernel thread for transmit. Also sets the TxThread bit of the Config register (see Section A.3.1).
rss	Bool	False	Enable receive side scaling. Sets the RSS bit of the Config register (see Section A.3.1).
rx_max_copy	MemorySize	1514B	Set the maximum number of bytes per RxData operation. Sets the RxMaxCopy device register (see Section A.3.5).
tx_max_copy	MemorySize	16kB	Set the maximum number of bytes per TxData operation. Sets the TxMaxCopy device register (see Section A.3.6).
rx_max_intr	UInt32	10	Set the maximum number of receives processed per interrupt. Sets the RxMaxIntr device register (see Section A.3.7).
rx_fifo_threshold	MemorySize	48kB	Set the receive FIFO low watermark value used to determine when to trigger an RxFifoMark interrupt (see Section A.3.3). Also sets the RxFifoMark device register (see Section A.3.10).
tx_fifo_threshold	MemorySize	16kB	Set the transmit FIFO high watermark value used to determine when to trigger an TxFifoMark interrupt (see Section A.3.3. Also sets the TxFifoMark device register (see Section A.3.11).
virtual_nic	Bool	False	Enable V-SINIC. Sets the Vnic bit of the Config register (see Section A.3.1).
zero_copy	Bool	False	Enable zero copy receive. Set the ZeroCopy bit of the Config register (see Section A.3.1).
delay_copy	Bool	False	Enable zero copy receive. Set the DelayCopy bit of the Config register (see Section A.3.1).
virtual_addr	Bool	False	Enable virtual addressing on the device. Sets the Vaddr bit of the Config register (see Section A.3.1).

Appendix B

Additional Figures

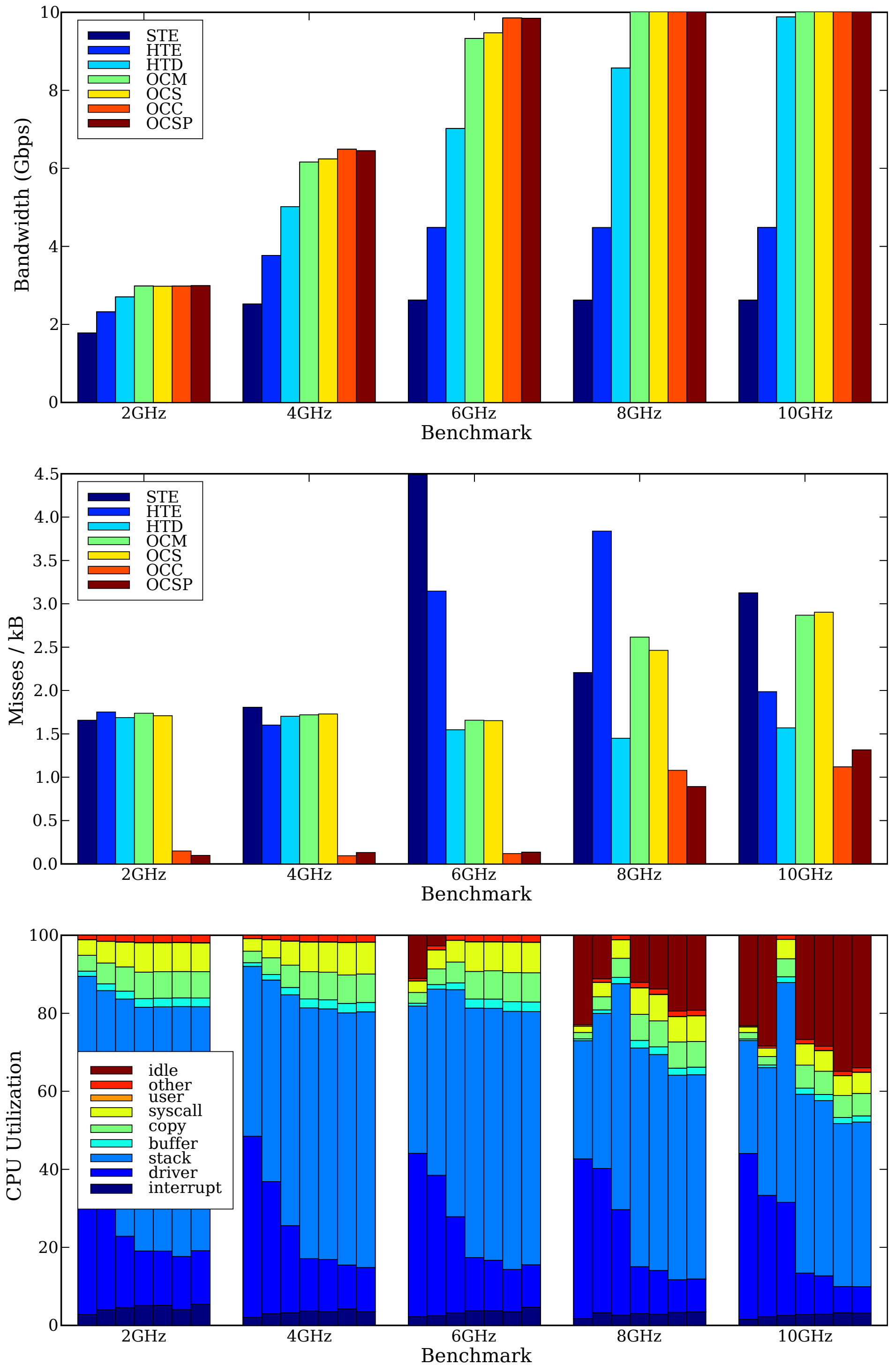


Figure B.1: TCP Transmit Micro-benchmark - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2GHz to 10GHz with a 16 MB level 2 cache size.

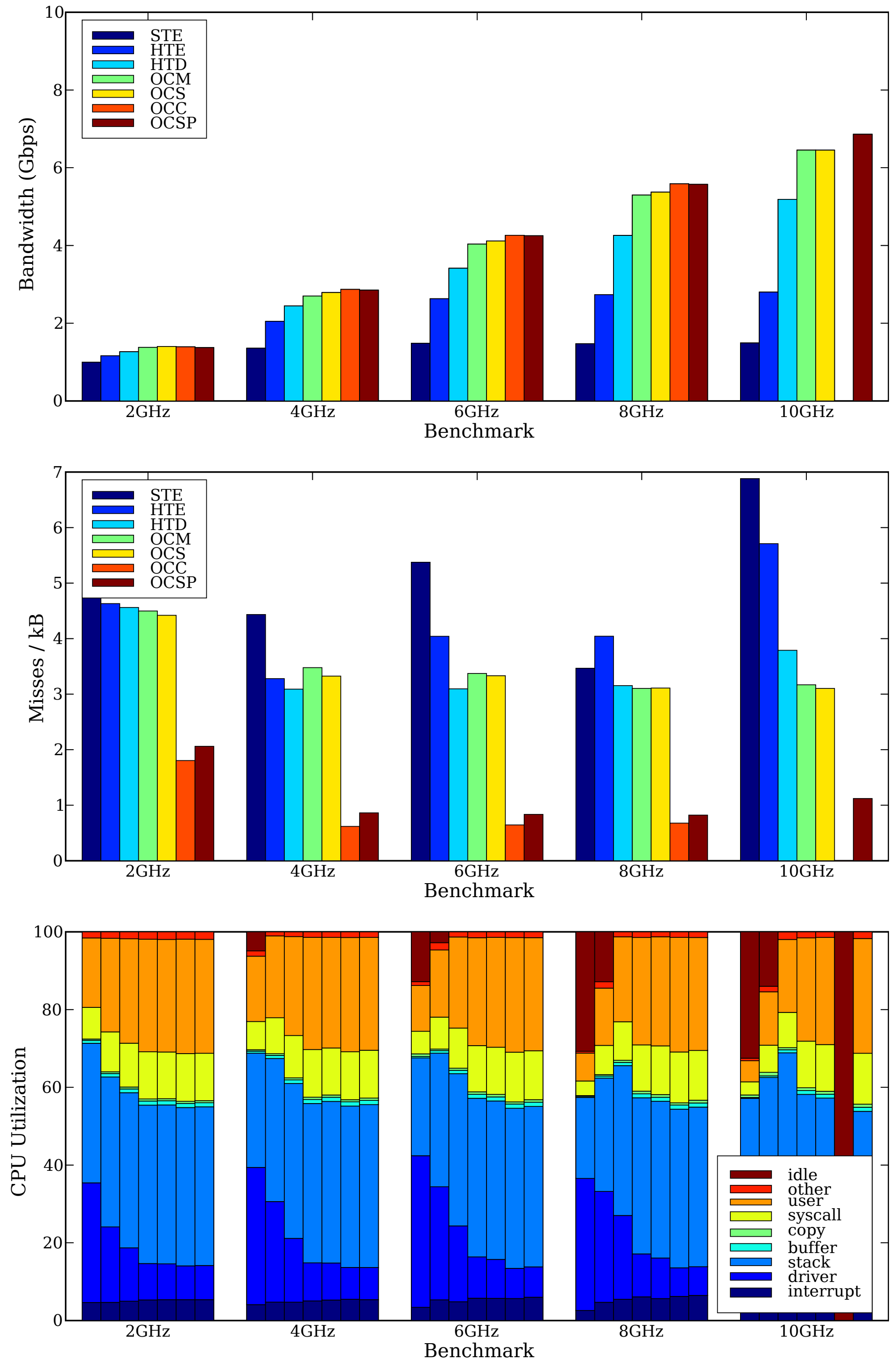


Figure B.2: Web Server Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2GHz to 10GHz with a 16 MB level 2 cache size.

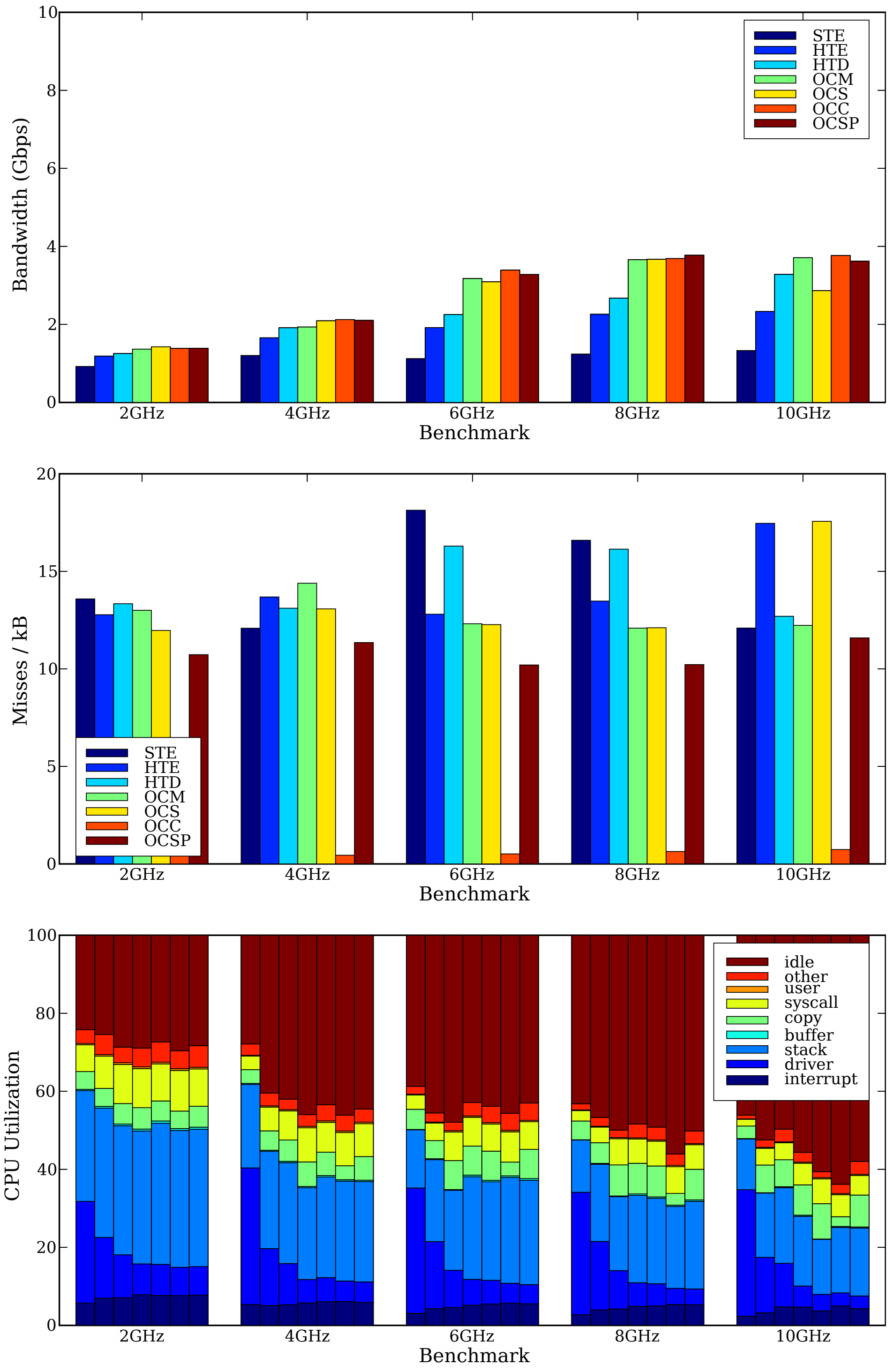


Figure B.3: iSCSI Initiator Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2GHz to 10GHz with a 16 MB level 2 cache size.

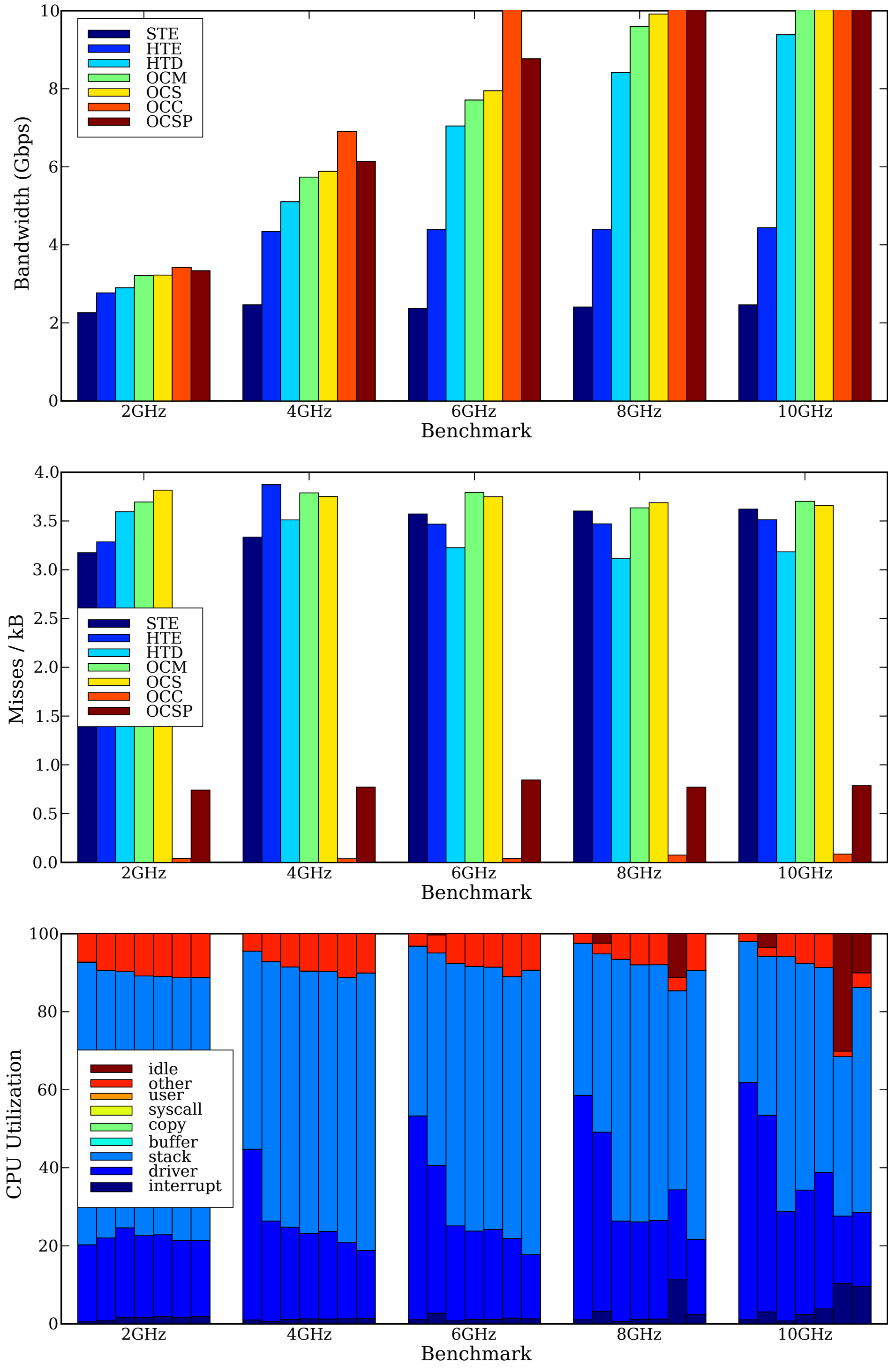


Figure B.4: Network Address Translation Gateway Results - The achieved bandwidth (top), Cache performance in number of misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the system configurations referred to in Figure 5.2 at CPU frequencies ranging from 2GHz to 10GHz with a 16 MB level 2 cache size.

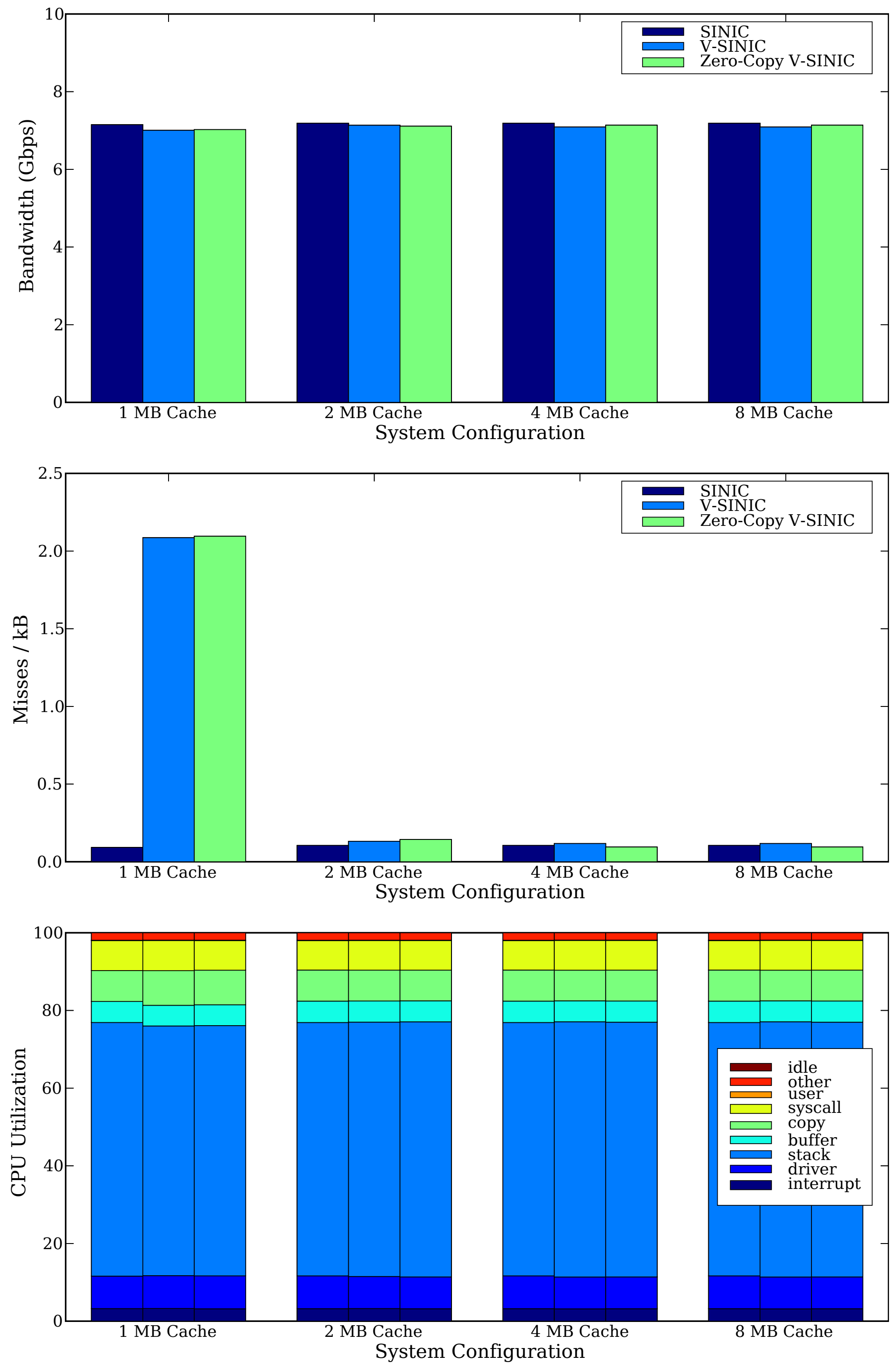


Figure B.5: Zero-Copy Transmit Results - Achieved bandwidth (top), cache miss rate in misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the transmit micro-benchmark with CPUs running at 4GHz.

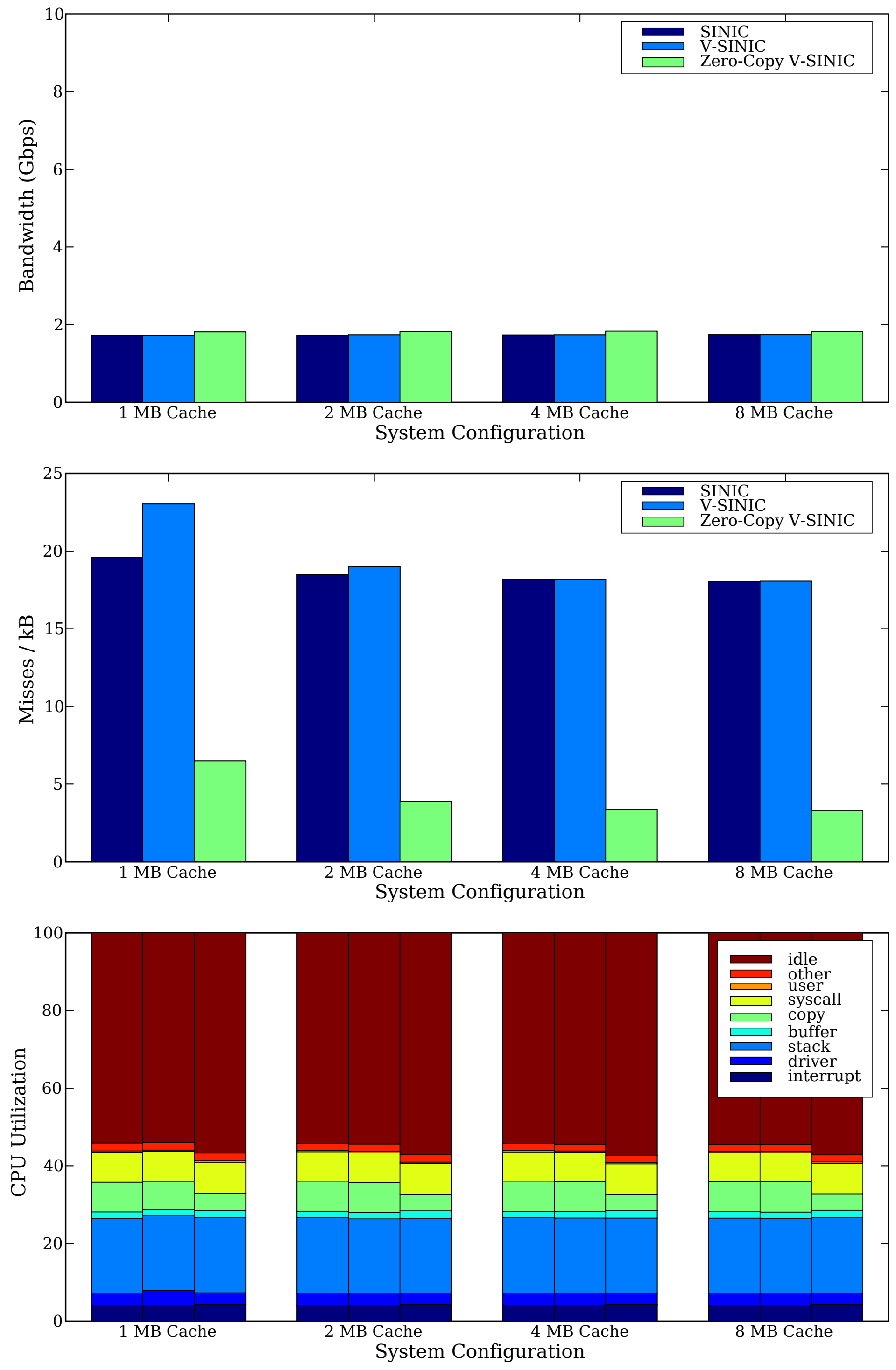


Figure B.6: Zero-Copy iSCSI Initiator Results - Achieved bandwidth (top), cache miss rate in misses per kilobyte of data transferred (middle), and CPU utilization breakdown (bottom) for the iscsi benchmark with CPUs running at 4GHz.

Bibliography

Bibliography

[1] Advanced Micro Devices, Inc. AMD eighth-generation processor architecture. White paper, October 2001. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Hammer_architecture_WP_2.pdf`.

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

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

[4] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proc. Seventeenth ACM Symp. on Operating System Principles (SOSP)*, volume 34, pages 232–246, December 1999.

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

[6] Ajay V. Bhatt. Creating a third generation I/O interconnect. `http://developer.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf`.

[7] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, Jul/Aug 2006.

[8] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Performance analysis of system overheads in TCP/IP workloads. In *Proc. 14th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 218–228, September 2005.

[9] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. 2nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 154–165, February 1996.

[10] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual memory mapped network interface for

the SHRIMP multicomputer. In *Proc. 21st Ann. Int'l Symp. on Computer Architecture*, pages 142–153, April 1994.

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

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

[13] 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.

[14] 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.

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

[16] Jeff Chase, Darrell Anderson, Andrew Gallatin, Alvin Lebeck, and Ken Yocum. Network i/o with trapeze. In *Proceedings of the 7th Hot Interconnects Symposium*, August 1999.

[17] 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.

[18] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, June 1989.

[19] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. SIGCOMM '90*, pages 200–208, September 1990.

[20] 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.

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

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

[23] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. Fourteenth ACM Symp. on Operating System Principles (SOSP)*, pages 189–202, December 1993.

[24] 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.

[25] D. Dunning, G. Regnier, eron D. Cam, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, Mar/Apr 1998.

[26] K. Egevang and P. Francis. RFC 1631: The ip network address translator (nat), May 1994. `http://www.ietf.org/rfc/rfc1631.txt`.

[27] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[28] 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.

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

[30] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev Patwardhan, and Greg J. Regnier. TCP performance re-visited. In *Proc. 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 70–79, March 2003.

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

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

[33] Thomas J. Hacker, Brian D. Athey, and Brian Noble. The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

[34] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proc. 27th Ann. Int'l Symp. on Computer Architecture*, pages 107–116, June 2000.

[35] 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.

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

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

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

[39] A. Huttunen, B. Swander, M. Stenberg, V. Volpe, and L. DiBurro. UDP encapsulation of IPsec packets. `http://www.ietf.org/internet-drafts/draft-ietf-ipsec-udp-encaps-07.txt`, October 2003.

[40] HyperTransport Consortium. HyperTransport technology - overview, 2003. `http://www.hypertransport.org/tech_overview.html`.

[41] InfiniBand Trade Association. About InfiniBand trade association: An InfiniBand technology overview. White paper, 2000. `http://www.infinibandta.org/ibta/`.

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

[43] Intel Corp. Interrupt moderation using intel gigabit ethernet controllers. White paper, September 2003. `http://www.intel.com/design/network/applnots/ap450.htm`.

[44] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proc. SIGCOMM '93*, pages 259–268, September 1993.

[45] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in TCP/IP. In *IEEE/ACM Transactions on Networking*, pages 817–828, 1996.

[46] S. Kent and R. Atkinson. RFC 2401: Security architecture for the internet protocol. `http://www.ietf.org/rfc/rfc2401.txt`, November 1998.

[47] 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.

[48] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[49] Xbit Laboratories. http://www.xbitlabs.com/articles/cpu/display/lga775_19.html.

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

[51] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proc. Fourth ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 272–285, July 1992.

[52] M5 Development Team. The M5 Simulator. http://m5.eecs.umich.edu.

[53] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Halberg, Johan Hogberg, Fredrik Larsson, Adreas Moestedt, , and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

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

[55] Microsoft. Scalable networking: Network protocol offload - introducing TCP chimney. White Paper. available at http://www.microsoft.com/whdc/.

[56] Microsoft. Scalable networking: Eliminating the receive processing bottleneck-introducing RSS. White Paper, April 2005. http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx.

[57] Jeffery C. Mogul. TCP offload is a dumb idea whose time has come. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, pages 25–30, May 2003.

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

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

[60] John Nagle. RFC 896: Congestion control in IP/TCP internetworks. http://www.ietf.org/rfc/rfc896.txt, January 1984.

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

[62] 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.

[63] David Oehmke, Nathan Binkert, Steven Reinhardt, and Trevor Mudge. How to fake 1000 registers. In *38th Ann. Int'l Symp. on Microarchitecture*, pages 7–18, November 2005.

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

[65] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Trans. Computer Systems*, 18(1):37–66, 2000.

[66] David C. Plummer. RFC 826: An Ethernet address resolution protocol. `http://www.ietf.org/rfc/rfc826.txt`, November 1982.

[67] Jon Postel. RFC 768: User datagram protocol. `http://www.ietf.org/rfc/rfc768.txt`, August 1980.

[68] Jon Postel. RFC 791: Internet protocol, September 1981. `http://www.ietf.org/rfc/rfc791.txt`.

[69] Jon Postel. RFC 793: Transmission control protocol. `http://www.ietf.org/rfc/rfc793.txt`, September 1981.

[70] R. Recio, P. Culley, D. Garcia, and J. Hilland. A rdma protocol specification, October 2003.

[71] 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.

[72] 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.

[73] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, Winter 1995.

[74] Ali G. Saidi, Nathan L. Binkert, Lisa R. Hsu, and Steven K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *Proc. 2005 Workshop on Interaction between Operating System and Computer Architecture (IOSCA)*, pages 33–38, October 2005.

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

[76] Lambert Schaelicke and Mike Parker. ML-RSIM reference manual. `http://www.cse.nd.edu/~lambert/pdf/ml-rsim.pdf`.

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

[78] W. Simpson. RFC 1661: The point-to-point protocol (PPP). `http://www.ietf.org/rfc/rfc1661.txt`, July 1994.

[79] W. Simpson. RFC 1853: Ip in ip tunneling. `http://www.ietf.org/rfc/rfc1853.txt`, October 1995.

[80] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to Gb/s networking. *IEEE Network*, pages 44–52, July 1993.

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

[82] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. `http://www.eetimes.com/story/OEG20040202S0004`.

[83] The Tolly Group. Tolly group report on etherfabric, July 2005. `http://www.level5networks.com/documents/TollyTS205125Level5NetworksEtherFabricEFI-21022TNICJuly2005.pdf`.

[84] 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.

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

[86] 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.

[87] 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.