RICE UNIVERSITY

Simulation-Driven Design of High-Performance Programmable Network Interface Cards

by

Paul Willmann

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Approved, Thesis Committee:

Vijay S. Pai , Chair
Assistant Professor of Electrical and Computer Engineering and Computer Science

---

Scott Rixner
Assistant Professor of Computer Science and Electrical and Computer Engineering

---

Bart Sinclair
Associate Dean of the George R. Brown School of Engineering

Houston, Texas

May, 2004

ABSTRACT

Simulation-Driven Design of High-Performance Programmable Network Interface Cards

by

Paul Willmann

As network link speeds race to 10 Gigabit/sec and beyond, Internet servers will rely on programmable network interface cards (NICs) to relieve the ever increasing frame processing burdens. To meet that need, this work introduces a scalable, programmable NIC architecture that saturates a full-duplex 10 Gigabit/sec Ethernet link. This proposed architecture utilizes simple parallel processors instead of a single complex core to satisfy its frame-processing requirements, thereby reducing core power by 63%. To exploit lower-frequency parallel resources, this work also contributes an enhanced event queue firmware mechanism that enables frame-level parallelism.

Although simulation provides a detailed, inexpensive method to evaluate architectures and software, no detailed architectural simulator has previously targeted NIC designs. This work therefore contributes Spinach, a new simulation toolset that accurately models programmable NICs in microarchitectural detail. A Spinach model

of an existing Gigabit NIC validates hardware benchmarks within 8.9% and yields solutions to previously undiscovered performance bottlenecks.

Acknowledgments

I would like to thank Dr. Vijay Pai and Dr. Scott Rixner for their motivation and guidance in this work. Also, I would like to thank Dr. Bart Sinclair for his perspectives on simulation approaches and methodology. Additionally, I want to acknowledge Mike Brogioli's significant technical and moral support throughout the development of this project. Many coffee-sustained nights (and following days) went into this work, and Mike worked through many of them with me. Hyong-youb Kim's expertise on all things related to the Tigon and its firmware proved invaluable toward my understanding in the development of this work. Neil Vachharajani provided extensive debugging of and improvements to the Liberty backend, for which I am very thankful. I also thank Dr. Jan Hewitt for sharing her vast wisdom regarding language and thesis semantics with me. Finally, I must thank my family, Leighann, and my friends. This work would not be possible without their steadfast support and understanding.

Contents

List of Figures

List of Tables

Chapter 1
Introduction: Designing High-Performance Programmable Network Interfaces Using Simulation

As computer architects strive to provide server solutions that meet the ever-growing demand for Internet services, the performance and capabilities of the server's interface to the network are becoming increasingly critical. Physical link speeds are increasing such that the server's host processor cannot keep up with the frame rates delivered by modern 10 Gigabit/sec links. Hence, next-generation network interface cards (NICs) will need to do more than just provide frames at the rate supported by the physical link; these NICs will also need to alleviate the frame-processing burdens currently carried by server host processors. Adding flexible computation capabilities to the NIC offers the potential to close the performance gap between the network and host processor and to improve overall server performance. However, as computer architects aim to close this gap using programmable NIC designs and specialized NIC firmware, architects will need a tool that can model their proposed NIC systems, can evaluate NIC performance, and can provide detailed information that will help improve their hardware and software designs. To meet the need for a modern, programmable NIC that can fully utilize next-generation 10 Gigabit/sec Ethernet links, this thesis contributes a multiprocessor programmable NIC architecture and paral-

lelized firmware. To develop this and future NIC designs, this work also introduces Spinach, a flexible simulation toolset for modeling programmable network interface architectures.

Although the advent of 10 Gb/sec Ethernet intensifies the need for programmable NICs, several researchers previously examined improving server performance by providing network services using programmable NICs. Gallatin et al. used a programmable Myrinet NIC designed for message-passing applications and developed firmware for it that improves TCP/IP performance [18]. Shivam et al. developed specialized firmware for the Tigon programmable Gigabit Ethernet NIC that improves message-passing application performance [43]. Kant estimates the web-serving benefits of offloading TCP/IP processing to an external accelerator, which could be part of a programmable NIC [23]. The 3Com corporation offers a line of programmable 100 Mbit/sec NICs that provide various offload capabilities, including IPSec firewall and cryptographic processing [1]. Intel produces a Gigabit NIC that uses an XScale processor to provide iSCSI services [22]. NIC programmability is even more important in light of the performance gap between 10 Gb/s Ethernet links and server host processor capabilities. This thesis extends the field of programmable NIC research by proposing a high-performance programmable NIC architecture and by introducing the Spinach toolset, which researchers can use to develop new programmable NIC designs and NIC firmware.

1.1 Next-Generation Network Performance via a Scalable, Programmable Ten Gigabit/sec NIC

The goal of a high-performance 10 Gb/s programmable NIC is to offer an extensible platform from which to provide frame processing services while ensuring that the NIC's frame rates match those delivered by 10 Gb/s Ethernet links. However, programmable NIC designs must also consider requirements beyond those defined simply by performance demands. Since a high-performance NIC is a server peripheral that typically resides in a data center, NIC designs must consider the tight power budget for peripheral devices operating in such an environment. While high-frequency, complex processor cores can deliver the high computation rates required to process frames at 10 Gb/s, such cores consume power roughly proportional to the square of their frequencies. Rather than relying on complex, high-frequency cores, the proposed architecture satisfies its computation requirements with several parallel, simple cores.

Firmware analysis shows that a programmable 10 Gb/s NIC must support 5.79 Gb/s of sustained control data bandwidth and at least 39.5 Gb/s of sustained frame data bandwidth. Delivering this substantial aggregate bandwidth motivates a memory organization that exploits the varying access behaviors of NIC data. The frame throughput effects resulting from I/O metadata latency encourage the use of low-latency memory structures; such structures are incompatible with the requirement for high-bandwidth frame data. We propose a heterogeneous, content-specific ap-

proach to satisfy the high-bandwidth and low-latency requirements simultaneously by partitioning the memory space according to data type. The proposed 10 Gb/s NIC architecture satisfies control data requirements by using a low-latency banked scratch pad memory network and satisfies frame data requirements by using off-chip high-bandwidth GDDR SDRAMS operating at 500 MHz.

Utilizing the proposed architecture's parallel cores requires an efficient, parallel firmware. NIC firmware interacts with the host through a series of steps that facilitate the transmission and reception of Ethernet frames. Existing parallelization approaches to programmable Gigabit NICs rely on coarse-grained parallelism among the various NIC processing steps. While such a parallelization method allows multiple steps to execute concurrently, each currently executing step must be unique. For example, only one instance of the step that processes frames arriving from the Ethernet can run at a time. In contrast, this work introduces an enhanced NIC firmware that exploits parallelism among frames and enables frame processing at the high rates provided by 10 Gb/s Ethernet. This frame-level parallelization permits multiple frames in the same step of NIC processing to proceed concurrently. An enhanced frame-level parallelized firmware enables a 10 Gb/s NIC with six 200 MHz parallel cores to achieve the same frame throughput performance as a single 800 MHz core while potentially using 63% less processor power than the single-core solution.

While our initial attempts at frame-level parallelized firmware produced correct

results, performance degraded as frame rates increased because of inefficiencies introduced by maintaining frame ordering. Maintaining frame ordering is important to avoid performance degradation associated with out-of-order TCP packet delivery. This thesis contributes firmware enhancements that led to a 213% improvement in overall frame throughput for minimum-sized frames while still maintaining frame ordering. NIC performance when processing minimum-sized frames is correlated to overall server performance [36].

1.2 Spinach: A Simulator for Programmable NIC Architectures

Analyzing the performance and runtime behaviors of a programmable NIC and its firmware requires a flexible infrastructure that permits detailed observation without perturbing system performance. Hardware implementations cannot provide such detailed insights, and they bear enormous manufacturing costs. However, simulation provides an inexpensive mechanism for iterative design evaluation. Simulation also provides valuable insights into hardware behaviors that are otherwise unobservable and offers more control than hardware for developing and debugging software.

Traditionally, the relevant complex systems in the computer architecture community include general purpose processors and multiprocessor systems. Hence, most simulation technology targets these types of systems. However, the advent of programmability in various embedded devices, including NICs, motivates a simulation

infrastructure that can accurately model the environment and input/output (I/O) interactions typical in such a device. While the applications for programmable NICs continue to grow, no simulator has yet targeted the NIC design space. This thesis contributes Spinach, a simulation toolset that accurately models programmable NICs in microarchitectural detail. Because host and network interactions arrive unsolicited with respect to the NIC's instruction flow, NICs differ from traditional processors. This uniqueness motivated a careful search of existing simulation techniques to select an appropriate method. The Liberty Simulation Environment (LSE) is a framework that enables straightforward mappings of parallel, asynchronous hardware components to simulation constructs.

Spinach is based on the LSE, in which users compose simulators hierarchically out of base modules. The LSE enables extensive code reuse and provides the port I/O abstraction, which permits tight coupling of timing and machine state. LSE modules evaluate their ports on a cycle-by-cycle basis, which provides the ability to transfer state and to service I/O interactions asynchronously with respect to instruction flow. Spinach goes beyond traditional LSE modules, however, by eliminating reliance on a backend emulator to maintain machine state; instead, Spinach modules maintain local state. Since Spinach modules can communicate only via their cycle-accurate ports, they closely map hardware structures to simulation modules. The elimination of reliance on an emulator means that users can freely reconfigure Spinach modules and be

certain that the reconfigured simulator will model their changes accurately. Furthermore, reconfiguration consists only of reconnecting and instantiating new modules; a user does not need to know anything about how a sequentially coded emulator models the simulated hardware state, since state changes are entirely encapsulated within the reconfigured modules. Spinach's low overall software engineering cost enables rapid and detailed examination of a wide NIC design space.

The Spinach library consists of 21 base modules from which users can build and evaluate NIC models. This work validates these Spinach modules by modeling a real hardware NIC and by comparing benchmarks. When modeling the existing Tigon-2 Gigabit NIC, Spinach predicts performance within 8.9% of hardware performance. Beyond simple validation, Spinach also enables exploration of modified architectures. Evaluating modified architectures led to the unanticipated discovery that the Tigon-2 Gigabit Ethernet NIC is both memory- and computation-bound when processing minimum-sized Ethernet frames. Adding a separate memory and bus to service I/O transfer metadata requests increases performance by almost 50% when processing miminum-sized frames.

This insight into components of frame throughput performance leads to the partitioned, heterogeneous memory organization of the proposed 10 Gb/s NIC. Furthermore, Spinach enables the firmware enhancements that improve performance relative to the initial firmware implementation for the proposed 10 Gb/s NIC. Spinach-

provided data builds an efficiency profile for the proposed NIC's parallelized firmware and provides detailed information that enables firmware optimizations that increase parallelism and decrease inefficiencies.

1.3 Introduction to Chapters

This thesis proceeds as follows: Chapter 2 gives background on programmable NIC functionality and the simulation underpinnings on which Spinach is based. Chapter 3 introduces Spinach in detail, including validation information and study of modified architectures. Chapter 4 introduces a new 10 Gb/s NIC architecture and evaluates its hardware and firmware effectiveness. Chapter 5 discusses related work, and Chapter 6 summarizes the contributions of this thesis.

Chapter 2
Background

This chapter provides background information relevant to network interface cards and simulation. Section 2.1 deals with NICs, including the tasks they perform and the firmware that implements those tasks. This section also introduces the Tigon-2 programmable Gigabit NIC architecture, which researchers have used as a platform from which to implement network services [25, 43]. Section 2.2 then presents information relevant to simulating programmable NIC systems, including an overview of the Liberty Simulation Environment on which Spinach is built.

2.1 Network Interface Cards

2.1.1 NIC Processing

The purpose of a network interface card is to facilitate the transmission and reception of Ethernet frames. User software typically requests network service, and the host operating system (OS) satisfies these requests by interacting with the NIC through a series of steps. This section describes those steps.

When frames arrive from the network, the network interface notifies the OS. Analogously, the OS notifies the network interface when frames are ready to be transmitted. After notification, the NIC and OS collaborate further to complete frame transfers.

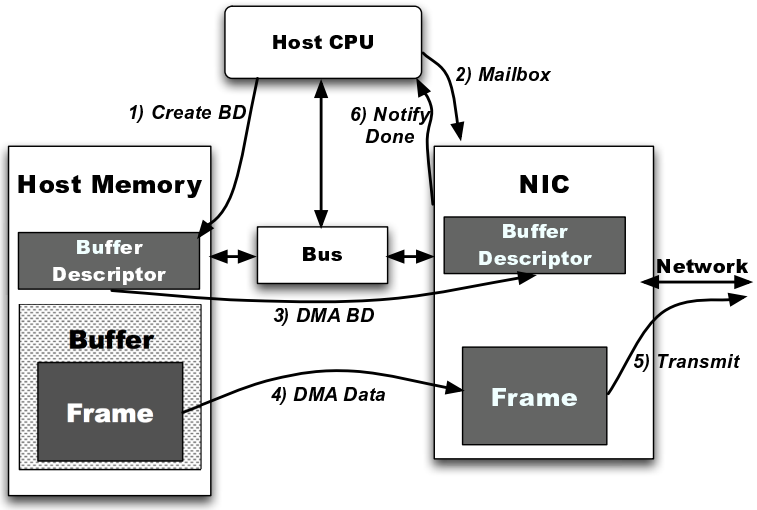


Figure 2.1 Steps for sending a frame with a NIC.

For both sending and receiving frames, the OS maintains buffers of free space that are allocated to frames. When the OS prepares to send a frame, it copies the frame contents to a series of free buffers and builds descriptors for the buffers that hold that frame. These descriptors, called *buffer descriptors*, contain the size and host memory address of the buffers. Likewise, the OS must preallocate free buffers so that the network interface has somewhere to copy frames as they arrive. These preallocated receive buffers also have buffer descriptors (BDs); receive BDs convey information about how much space is available in host memory for frame delivery. The OS organizes both send and receive BDs in circular rings, each of which has a producer and consumer value that indicate the amount of free BDs in each ring.

Figure 2.1 shows the steps involved in sending a frame from the host to the network

via a NIC. It is assumed that before these steps take place, the OS has copied the frame into a free buffer and is otherwise ready to send the frame. From there, the processor *1)* fills in the contents of a buffer descriptor corresponding to the frame that is to be sent. The specific BD filled in is the next available in the send BD ring, indicated by the send-buffer-descriptor producer pointer. To indicate that a frame is ready for transmission, the OS *2)* writes the updated send-buffer-descriptor producer value into the NIC via programmed I/O. As indicated by the figure, these updates to the buffer-descriptor producer are sometimes referred to as mailbox writes. The NIC maintains its own copy of the host's send-buffer-descriptor ring and a local producer value, and thus the NIC can determine when new BDs are available. After being notified of the updated send buffer producer value, the NIC *3)* reads the host's BD into local NIC memory via direct memory access (DMA). When the send BD arrives, the NIC analyzes the buffer and *4)* fetches the associated host buffer. Once the frame data has fully arrived, the NIC *5)* transmits the frame out over the network using the medium access control (MAC) unit. Once the transmit has completed, the NIC may *6)* notify the OS by interrupting the host's processor. An interrupt coalescing threshold set by the OS determines whether or not the NIC will interrupt the host. This value specifies how many frames must finish transmission before the NIC interrupts the host, which coalesces several events into one interrupt and hence reduces host interrupt overhead.

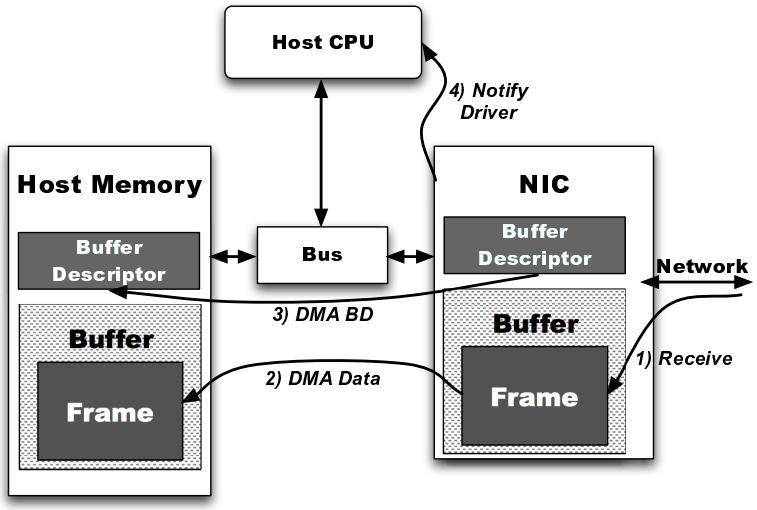


Figure 2.2 Steps for receiving a frame with a NIC.

Figure 2.2 presents the steps required to receive a frame from the NIC into host memory. The initial conditions for receiving a frame are slightly more complex than for sending one. The steps in this figure begin after the OS has already created free receive buffers and corresponding BDs in host memory. Furthermore, the host has already notified the NIC of the existence of these free buffers, and the NIC has read the corresponding BDs into NIC memory via DMA. Notification of free receive buffers is done analogously to the notification of available send buffers.

Once free buffers are available, the NIC *1)* receives network data into local NIC memory. After the frame completes its arrival, the NIC *2)* transfers the frame contents via DMA to the host memory location pointed to by an available receive BD. After that, the NIC *3)* modifies the NIC's copy of the corresponding receive BD's length

field in order to indicate the size of the received frame and then writes the modified BD to host memory via DMA. Finally, the NIC *4)* notifies the OS that a frame has arrived. Typically this notification comes in the form of a DMA write of the newly updated receive-buffer-descriptor consumer pointer, which indicates that the NIC has consumed free buffers. However, this notification may also be an interrupt. Both forms are subject to coalescing thresholds, just as in the send case.

These steps are somewhat simplified for clarity. In practice, frames may be transferred between the NIC and host in the form of several buffers and BDs, rather than just one each as depicted here. Commonly, received frames use just one buffer and descriptor pair (presuming that the allocated buffers are of large enough grain to hold the received frame), while sent frames typically use two separate buffer and descriptor pairs for a frame's IP header and its payload data. Additionally, the operating system may or may not actually copy data between user- and kernel-space buffers. To avoid the memory overhead of copying, the OS may simply remap the virtual pages from user to kernel space or out of kernel space into user space [16]. Regardless of these differences, the NIC's tasks and OS's responsibilities are identical.

2.1.2 The Tigon-2 Architecture

Since the NIC processing steps presented in Section 2.1.1 are straightforward and methodical, most NICs manage these steps with high-speed application-specific integrated circuits (ASICs). However, programmable architectures offer flexibility to

redefine the steps performed by the NIC and add functionality into those steps. This functionality enables offloading portions of server computation to the NIC. As mentioned in the introduction, several researchers have perviously examined offload capabilities of programmable NICs. Existing programmable NIC capabilities include iSCSI offloading, IPSec firewalling and cryptography, message passing, TCP checksumming, and data caching [22, 1, 43, 28, 25]. The Tigon-2 NIC architecture is the only known user-programmable Gigabit Ethernet NIC architecture and is the basis of the AceNIC family of 3Com Gigabit NICs. When Alteon Websystems released the firmware to the Tigon-2 as open-source software, it also released the hardware and firmware manuals which describe the Tigon-2 hardware and software architectures [5, 6]. This section uses that information to summarize the Tigon-2.

The Tigon-2 architecture has two processors running at 88 MHz, one private scratch pad memory per processor, two DMA channels that interact with a peripheral component interconnect (PCI) interface, and a MAC unit to assist in transmitting and receiving frames over the full-duplex Gigabit Ethernet interface. Figure 2.3 depicts this architecture. The processors support a subset of the MIPS R4000 instruction set and two additional instructions for priority decoding and jump-table offset control flow. The 64-bit memory bus runs at 100 MHz. The connected external SRAM operates in burst mode with an initial latency of 2 memory cycles and a burst length of 4 64-bit words, providing a maximum achievable throughput of 5.12 Gb/sec. The

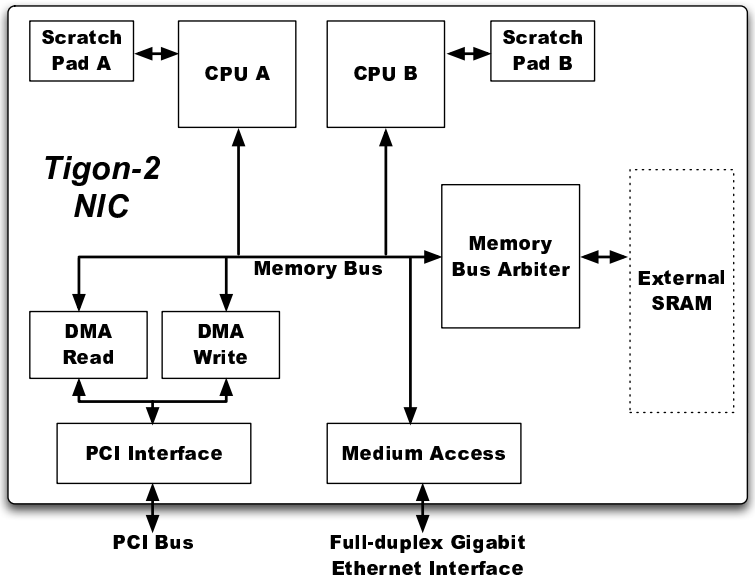


Figure 2.3 The Tigon-2 Gigabit programmable NIC architecture.

scratch pads provide access of up to one 32-bit word per core cycle. The processors and hardware assists also have single-cycle access to a set of memory-mapped registers.

The processors may use scratch pad memory for either instructions or data, but each scratch pad has only one port. Each processor also has a small hardware-controlled instruction cache that caches contents only from main memory (not the scratch pad.) These instruction caches each have a single 128-byte line for prefetching firmware not found in the scratch pad. The Tigon-2 uses a prioritized, burst-detecting bus arbitration scheme with timeouts to prevent starvation. Both processors share one arbiter request slot with round-robin arbitration between the processors, though the processors have the lowest prioritization among all requesting devices.

The hardware assists may be logically decomposed into four interfaces: DMA read, DMA write, MAC transmit, and MAC receive. DMA read transfers data from host memory to NIC memory, while DMA write transfers data in the opposite direction. Likewise, the MAC transmit interface sends frames from NIC memory out to the network, while the MAC receive interface writes received frames into NIC memory. These DMA and MAC assist units are simple nonprogrammable state machines that perform I/O services. The assists read transaction descriptors created by the processors and perform the requested operation. These transaction descriptors are in local NIC memory and are organized in circular descriptor rings, just like the buffer descriptor rings. Each ring has a producer and consumer value, and there are separate rings for the four assist interfaces. When the processors request service, they create a transaction descriptor in the next free slot of the appropriate ring and update the producer value. The assists likewise poll this producer value in order to determine when new requests arrive. As the assists complete transactions, they update the consumer value for their ring, indicating to the processor when the transactions have completed. The caveat to this description is that MAC receive works in reverse. The MAC receive interface produces transaction descriptors for received frames and writes the producer value, while the processors read these descriptors and update the consumer value as they process the received frames.

2.1.3 NIC Firmware

Though the hardware described in Section 2.1.2 facilitates the input and output (I/O) transactions required by a NIC, a programmable NIC's firmware must orchestrate the hardware in an efficient manner. Efficiency ensures timely completion of the NIC's processing steps. This section describes the Event Model abstraction of NIC processing and a mapping of event processing to hardware resources.

The Event Model

Most steps described in Section 2.1.1 correlate to actions that the NIC must take to further the processing of a frame. These actions taken are called *events*, and either hardware or software may trigger them. The firmware functions that service events are called *event handlers*. Hardware events result when the hardware completes transactions. For instance, when the DMA read interface finishes processing transactions, a DMA read event occurs and the DMA read handler services that event. On the other hand, software events are the result of previous events that may need further, separate service. For example, during the process of servicing a DMA read event, the DMA read handler inspects the type of data that has been read. This is done at the completion of step *3)* in Figure 2.1. If the data is a group of send BDs, the DMA read handler will trigger a separate software event for the purpose of requesting the frames associated with these newly received send BDs. This requesting process is

the initialization of the read transaction represented by step *4)* in Figure 2.1. Correctly triggering and clearing these various events as each step proceeds requires a management mechanism.

The Tigon-2 architecture manages events via a hardware event register mechanism, while the Tigon-2's specialized instructions provide efficient event dispatch. Each processor has its own event register. Event registers are bit vectors in which each bit represents a distinct event. Events are prioritized by the rank of their bit position in the event registers. Processors use the specialized priority-decode instruction to determine the highest-order event currently set. The result of this priority decode feeds the specialized jump-offset instruction, which transfers control via a jump table to the event's handler.

As hardware events complete, the Tigon-2 automatically sets a bit in the event registers associated with that type of event. For instance, the hardware controls DMA read, DMA write, MAC transmit, and MAC receive events. These hardware events are subject to a software-controlled threshold which sets how many transactions must finish before the next event should be triggered. The Tigon-2's processors may write to the event register in order to clear events or set software events.

Task-level Parallelism

As Figures 2.1 and 2.2 show, many steps (and thus events) are involved in sending and receiving frames. To maximize frame throughput, event processing should be

overlapped in such a way as to allow multiple frames to be in various stages of processing at any given time. Alteon's Tigon-2 firmware (version 12.4.13) processes all event handlers on one processor while executing the timer event handler on the other [6]. Kim et al. examined the data sharing patterns and load balancing among the Tigon-2 event handlers and repartitioned them such that send-related handlers ran on one processor while receive-related handlers ran on the other, resulting in a 65% performance increase [26]. While the Alteon firmware exploits very little parallelism, both approaches rely on *task-level* parallelism. In the programmable NIC domain, task-level parallelism refers to the concurrent execution of parallel event handlers. A task is one event handler that implements a NIC processing step.

Figure 2.4 illustrates task-level parallelism using an event register; double-black arrows indicate bits being set, double-clear arrows represent bits being cleared, and single arrows represent event dispatch. In this case, the hardware sets the DMA-read event bit in the event registers. A processor assigned to this type of event then runs the DMA-read handler (labeled "Process DMA Read"), which inspects the completed DMA-read transactions. As shown in the figure, when the handler encounters receive and send BDs, the DMA-read handler turns on the respective software event bits in the event register. Assuming there are sufficient processors, each of these types of handlers (receive-BD and send-BD handlers) may run concurrently after the DMA-read handler toggles their event bits. Kim's parallelization strategy is such that

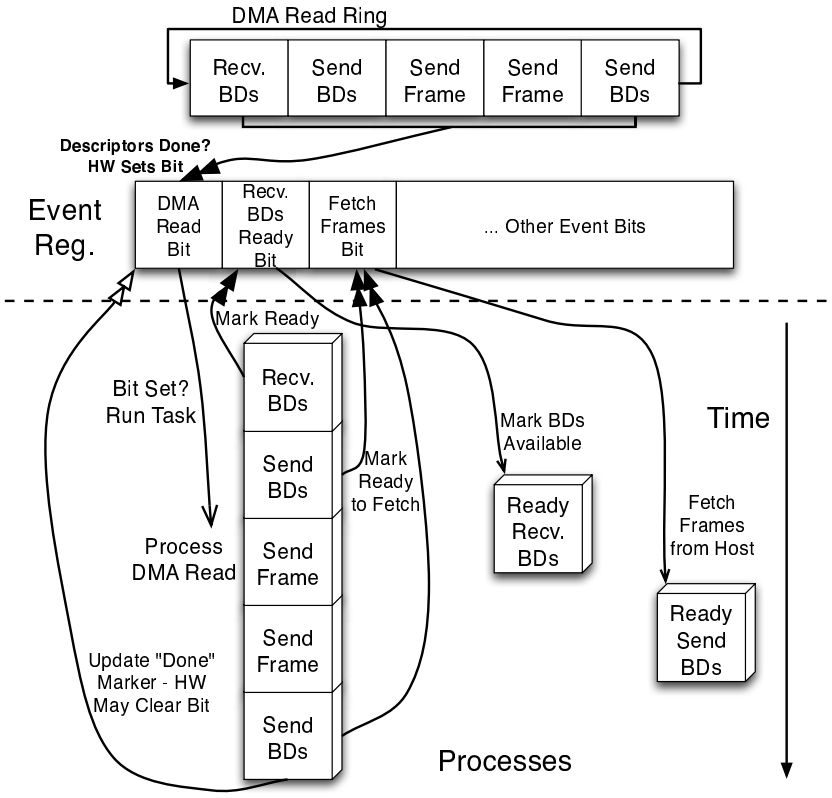


Figure 2.4 Task-level parallelism using an event register.

the "Receive-BDs ready" event handler could run concurrently with the DMA-read handler, but the "Fetch Frames" handler would have to run after the DMA-read handler completes, since it is statically assigned to the same processor as the DMA-read handler.

Regardless of partitioning issues, the key observation regarding task-level parallelism is that event handling occurs at the granularity of per-type tasks. With an event register, there is no notion of a partial event; events are either on (some data is ready

to be processed) or off (no data available for processing at this step). For instance, the DMA-read handler in Figure 2.4 processes all currently completed DMA-read transactions. Task-level parallelism's scalability depends upon the handlers having relatively equal computation demands and experiencing low data-sharing costs where data sharing exists. However, Kim's profiling work shows that the Tigon-2 handlers are not equally balanced [26]. Furthermore, the Tigon-2 architecture only supports interprocessor data sharing through the main memory, for which the processors have the lowest arbitration priority. Using the task-level parallelism approach with more than two processors on an architecture similar to Tigon-2 is thus unlikely to scale efficiently.

2.2 Simulation

Programmable NIC systems, such as those described in the preceding section, represent an interesting area of research because such NICs will enable network servers to utilize the frame rates delivered by next-generation links. When developing other programmable systems, such as uniprocessors and multiprocessors, researchers utilize simulation as a mechanism to develop and enhance their designs.

Simulators are useful tools that system designers utilize in various stages of the design process. However, the flexibility and accuracy of a simulator often determine its utility in design. Simulators derived from transistor-level or low-level register transfer language (RTL) descriptions are highly accurate and can yield important, physically

unobservable insights into performance behaviors and bottlenecks. The Intel IXP simulator falls in this category [7]. Such simulators result from a specification of a highly detailed design that has already progressed far beyond the point of microarchitectural exploration. This class of simulator is useful for modeling low-level details of an existing, established design. However, they tend to be poor tools for designing new systems. For instance, if designers propose an entirely new memory hierarchy, at minimum this requires a complete rewrite of that subsystem including low-level implementation details. Portions of the system that interact with the memory hierarchy may also require modification.

Another approach to simulation is to model high-level architectural constructs in a modular fashion. Computer architects then organize the overall model using a combination of these high-level pieces. This simulation approach provides flexibility, but that flexibility can come at the price of accuracy. Recent trends in architectural simulation favor detailed approaches in order to reduce accuracy concerns. Pai et al. show that accurately modeling ILP mechanisms in detail significantly improves the accuracy of ILP shared-memory multiprocessors [34]. In that case, modeling the processor interconnect network and memory system in detail may prove useless and misleading if the processor models do not accurately model memory interactions.

To be useful from the beginning to end of the design process, a simulator should be flexible enough to permit rapid evaluation of a wide space of high-level design changes,

yet accurate enough to yield meaningful results. Several simulation frameworks in the domain of general-purpose processors attempt to meet these goals. Section 2.2.1 discusses existing simulator frameworks in light of the characteristics of NICs, and Section 2.2.2 discusses the LSE, on which Spinach is based. Section 5.1 further details modern multiprocessor and embedded processor simulators.

2.2.1 Evaluating Simulation Approaches for Suitability in the NIC Domain

The vast majority of architectural simulators are either discrete-event simulators or process-oriented simulators that result from machine descriptions. While these simulators may be modular, they are inherently sequential. Vachharajani et al. discuss the difficulty of correctly mapping parallel hardware operations to sequentially programmed simulators [45]. In particular, behavior implemented by module application programming interfaces (APIs) may have very little correlation to hardware behavior. Discrete-event simulators must take special care to model parallel hardware actions with sequentially processed events. Similarly, architectural machine description simulators require a careful mapping of potentially concurrent hardware actions to sequential simulator states.

While it may be straightforward to implement the functions internal to module APIs correctly, managing the method in which APIs are called complicates matters further. Module APIs can be called globally from any module to any other module,

and this can introduce correctness problems as designs evolve away from the original baseline configuration. Furthermore, a user has no guarantee that an added module will correctly affect the rest of the entire system unless that user fully understands the entire codebase. For instance, a new cache controller module may attempt to throttle bandwidth by taking more cycles to satisfy a request, but a calling module may have been written with an existing notion of what the bandwidth behavior used to be. The original calling modules may issue cache requests but then obtain the requisite values directly from global shared state after some internally set number of cycles. Subsequent calls may do the same thing. Thus the simulation proceeds using the new cache module and keeps getting correct values, but it is not readily apparent that timing is completely inaccurate. While this is completely a software engineering concern, the complications of managing such issues in large simulators with many developers are nontrivial. Consequently, taking an existing simulator codebase that functions in this way, such as SimpleScalar [10], is less than ideal because it limits flexibility and raises serious correctness and maintenance concerns.

The asynchronous interactions of a programmable NIC's assists and multiple processors further complicate matters with traditional discrete-event simulators. Such simulators schedule events for known times; when no events are scheduled, the simulation proceeds "instantly" over these empty cycles. However, NICs regularly process network and host requests that arrive unsolicited and asynchronously. Correctly

modeling this unsolicited behavior is essential to correctness because the timing of instruction execution and state modifications may have an impact on the actual instruction sequences computed [29].

Recent simulator frameworks attempt to build architectural simulators that better match the intended hardware. The Asim simulation framework models the execution of Alpha processors using a combination of a performance model that maintains timing and a backend execution layer that maintains machine state [17]. As instructions proceed through the performance model, the performance model's simulation units interact with the execution layer to update machine state. Modules within the performance model interact with each other and the execution layer via a set of APIs. Again, the ability to call a global API from anywhere in the code raises maintenance and correctness concerns.

While Asim better maps architectural simulation to hardware structures, it requires close integration with the execution layer, which is still a sequentially programmed emulator. If users modify the modular timing layer to represent a radically different design, the execution layer may not model the associated state transitions correctly. Straying from the state transition steps supported by the execution layer would require modification to the execution layer, which most likely would be a nontrivial software engineering effort. Additionally, the execution layer may obscure intercycle details relevant to the correctness of the system model.

Asim's performance model could correctly model the asynchronous, unsolicited requests from NIC traffic by having additional NIC-specific modules that call the memory modules for service. Thus, a framework such as Asim begins to address some of the concerns for flexible simulation of systems that do not closely match the timing characteristics of baseline configurations or that depend on interactions asynchronous to the flow of instructions, such as NICs. However, maintaining correctness with respect to the execution layer involves the same problems of traditional sequentially coded simulators. Specifically, the user must still call global APIs in a method consistent with timing, which necessarily implies a significant software engineering cost in large systems.

2.2.2 The Liberty Simulation Environment

Vachharajani et al. introduced the Liberty Simulation Environment (LSE) as a way to map hardware structures faithfully to simulation modules [45]. Their hierarchical method of building simulators that can tightly couple timing and simulation state is novel and still relatively new to the architecture community. This subsection compares the LSE to Asim and explains how computer architects may use the LSE to build useful simulators. The hierarchical LSE method enables code reuse that contributes to Spinach's flexibility. Spinach extends the LSE by tightly coupling machine state to each module rather than relying on the LSE's functional emulator.

The LSE is a framework and module set intended to explore microarchitectural

design choices rapidly. Computer architects compose LSE simulators using a set of these modules. Modules communicate with each other via *ports*. Users instantiate modules, connect the module ports, and compose hierarchical modules using a high-level language, the Liberty Structural Specification (LSS) [44]. The key to LSE simulators is the port construct; all communication between modules must take place between ports. The abstraction of a port-to-port connection is called a *channel*. Since all communication happens over channels, LSE modules have no notion of global APIs for communication. Instead, modules must convey communication via their output ports using "send" calls. However, each send call is specific to the port instance on which it is called, and a module may only call send or receive methods on local port instances. Modules cannot see a global port namespace.

Fundamentally, the modular Asim approach differs from the LSE approach in that LSE forces all communication across local module port instances, and the timing model used in LSE is directly coupled to dataflow. Forcing communication over enumerated ports ensures a close mapping of modules to hardware behavior, while the coupling of timing to dataflow enforces that data flows through the model as it would in the desired hardware. This coupling means that there is no potential disconnect between machine state timing and performance model timing which could affect correctness.

LSE simulators couple timing and state logically via interport channel connec-

tions. LSE channels support sending and receiving one unit of data per simulation cycle. Transmitting more requires more channel connections between the two communicating modules, just as increased per-cycle bus bandwidth requires more bits on the bus. Data units transmitted via ports are typically restricted to C's simple built-in types, but future versions of the LSE will support complex data types.

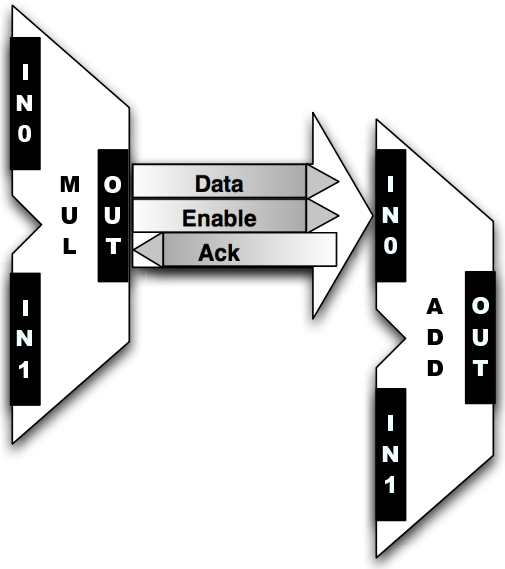


Figure 2.5 Example of an LSE channel connecting two ALUs.

LSE channels contain three parts: data, enable, and acknowledgement (Ack). Figure 2.5 depicts an LSE channel connecting the "OUT" port of one hypothetical MUL module to the "IN0" port of an ADD module. (In the interest of simplicity, full port connections are not shown.) After the MUL unit has its source operands, it produces its output and sends it out via its OUT port. This sets the data portion of the chan-

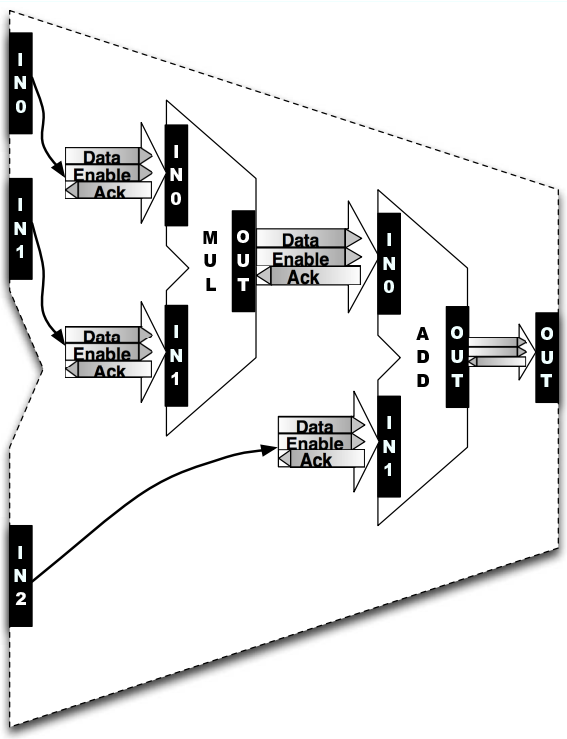


Figure 2.6 A hierarchical multiply-add LSE module.

nel. The LSE framework consequently schedules the ADD module to run, which sees the new input and may produce an output on its OUT port. The ADD module is responsible for passing back an Ack value to the MUL unit's OUT port. Typically LSE modules pass back the Ack supplied at their output to their inputs. This backward flow enables source modules in a chain can determine if subsequent dependent operations have been acknowledged. In this case, the ADD unit would pass whatever Ack value is present at its OUT port back to its IN0 and IN1 ports. This, in turn, arrives via the channel at the feeding modules, including MUL. Enables flow through

the system analogously, though in the opposite direction. Notice, however, that the enable portion of an LSE channel is largely redundant because LSE channels support sending "null" data, which effectively represents a disabled signal. The modules must facilitate all data, enable, and Ack flow explicitly. However, they can control only the channels that flow through them via their ports.

Figure 2.5 could be used as a building block for a multiply add (MADD) module. The LSS standard supports building modules hierarchically. Figure 2.5 depicts two low-level modules. Low-level LSE modules are written in C and use LSE port APIs to communicate; high-level LSE modules are composed of a mixture of low-level modules and other high-level modules. For instance, a user could define a new MADD module based on the two low-level ALUs, as in Figure 2.6. The newly defined MADD unit has three input ports (IN0, IN1, and IN2) and one output port (OUT) that would be defined in its LSS file. The MADD LSS specification would instantiate two ALUs (one parameterized to multiply and the other to add). The high-level MADD specification would then connect port instances of the component low-level ALUs by using LSS connection statements. After defining MADD, a user could instantiate it and use it interchangeably with any other high- or low-level module. When connecting modules in LSS, there is no difference between the two. LSE simulators use module blocks built similarly to the MADD example in order to construct architectural models.

The LSE's hierarchical code reuse is especially useful in creating multiprocessor

systems and memory organizations. Once a user creates a processor model, it is trivial to instantiate and connect multiple instances of that model. Furthermore, the LSE's tight dataflow/timing coupling ensures reliability when dealing with requests that are unsolicited and asynchronous to instruction execution. Such interactions may be requested by special-purpose modules that plug in (via ports) to the rest of the general-purpose execution and memory systems. These requests may arrive asynchronously and unsolicited, and will be dealt with cycle-by-cycle just as "normal" interactions from the processors would. Code reuse and the straightforward, correct mapping of hardware structures to simulator modules make the LSE a good candidate framework for simulating programmable NICs.

As with most simulators, LSE modules can take ordinary values as parameters. Unlike most other simulators, modules can also take user-written functions as parameters. These functions can preprocess input data and set the cycle-by-cycle behavior of the module accordingly. For example, bus arbiter units take an algorithmic parameter that determines which requesting unit can access the bus. This parameter effectively sets the cycle-by-cycle arbitration policy. Users can manipulate such algorithmic parameters to get very different behaviors using the same modules. Thus, key advantages of the LSE include the straightforward retargetability and reconfiguration of modules into simulators of various systems as well as the easy integration of various simulators with each other. The latter is particularly straightforward because even a

top-level simulator is itself an LSE module.

Chapter 3
Spinach: A Simulator for Programmable NIC Architectures

Because network interface cards can have a tremendous impact on server application performance, the design of efficient, high-performance NICs will be of critical importance as computer architects attempt to meet the increasing demands of next-generation applications. Offloading services normally provided by the server to the server's programmable NIC can improve performance [28]. TCP protocol processing, iSCSI services, and message passing services are surfacing as future applications for programmable NICs [21, 31, 43].

Simulation provides an inexpensive design tool that can reveal an arbitrary degree of detail and is widely used in general-purpose processor design. However, unlike general-purpose CPUs, NICs must handle host and network interactions that arrive and complete asynchronously with respect to the flow of instructions. This characteristic of programmable NICs makes most general-purpose CPU simulators poor candidates for extension into the domain of NIC modeling.

This chapter contributes Spinach, a flexible simulation toolset based on the Liberty Simulation Environment (LSE) framework. Spinach is the first known publicly available programmable NIC simulation toolset capable of modeling NICs in microarchitectural detail. Spinach introduces a new model of simulator computation ab-

straction that enables high degrees of configurability and requires very little software engineering overhead when using or recomposing Spinach modules. Spinach's configurability is limited only by the LSE framework's ability to produce and compile large, complex software systems that may result from complex Spinach definitions; this limitation is not inherent to Spinach.

Spinach is composed of only 21 low-level computation and state modules. Users may compose basic simulator computation and state modules into blocks; these blocks and basic modules may be composed hierarchically in any fashion the user wishes. However, existing LSE-based modules rely on an underlying emulator to maintain the architectural state of the system being simulated. An emulator backend such as this may introduce correctness problems when dealing with the asynchronous interactions prevalent in NIC processing. Furthermore, any change to the many modules which may affect the architectural state may require complementary changes to the emulator backend. Such changes may be subtle, thus increasing the likelihood of user error and the introduction of inaccuracies into the simulation model. Furthermore, correctly maintaining the backend emulator requires significant knowledge of how the backend emulator models its architectural state, which may be disjoint from the user's desired model. Spinach obviates all of these concerns by eliminating reliance on the emulator backend and instead maintaining the architectural state locally per module. Thus, Spinach modules provide a one-to-one mapping of hardware constructs and their

associated state (e.g., arithmetic units, memory units, etc.) to software modules. The straightforward architectural organization of Spinach simulators combined with the limitless flexibility of Spinach modules enables rapid evaluation of wide design spaces. For instance, modifying a single-processor NIC architecture to support an additional processor required only 90 lines of high-level script.

Section 3.1 of this chapter discusses the design goals for Spinach and the various simulation approaches available to meet those goals. Section 3.2 introduces the modules Spinach provides. Section 3.3 presents the Tigon-2 programmable NIC architecture and examines how Spinach modules map to this architecture. Section 3.4 evaluates the accuracy of Spinach modeling Tigon-2, finding that the Spinach model is within 4.5% of hardware performance on average. Section 3.5 concludes by evaluating some modified versions of the Tigon-2 architecture and discusses the importance of certain architectural parameters with respect to NIC processing. Portions of this chapter, including some figures, are based on a paper copyrighted by the Association for Computing Machinery [48] and are used in compliance with the ACM's authorship copyright agreement.

3.1 Simulating NICs

As described in Section 2.2, a large portion of previous simulation work is unsuitable as a basis for simulating NICs because of the unique characteristics NICs have, such as self-modifying code and I/O interactions that occur asynchronously to the

flow of instructions. Furthermore, many previous simulation frameworks present a significant software engineering challenge to computer architects because they require accurately mapping parallel hardware events to sequential simulator code. Global simulation application programming interfaces (APIs) can introduce artificial behaviors that model physical components poorly and can detrimentally affect correctness.

The Liberty Simulation Environment (LSE) attempts to address these issues. Designers build simulators by composing LSE modules hierarchically. Unlike simulators that allow global APIs for communication, LSE modules can communicate with each other only by their ports; modules evaluate ports on a cycle-by-cycle basis. The port construct encourages a close mapping of parallel hardware structures to LSE modules. Designers instantiate modules and connect module ports using the high-level Liberty Structural Specification (LSS) language [44].

3.1.1 Spinach: Going Beyond LSE

Ideally, reconnecting the component modules in different ways should change the execution and timing characteristics of the resultant simulator in a manner exactly representative of the module reorganization that the user intended. As previously described in Section 2.2.2, this is precisely what happens. However, as alluded to in that section, the caveat to flexible LSE module reorganization is the LSE's *emulator* interface. Though not discussed in the paper by Vachharajani et al. [45], the emulator is a fast functional simulator that maintains the architectural state of the entire

system. Thus, existing LSE simulators use the modules only for timing. The modules may send data, but most parts of existing LSE simulators do not use the dataflow to send and receive machine state (such as register values, memory data, and so on.) These simulators use the control flow aspects of the LSE port construct to maintain timing, and the modules internally call emulator APIs to manipulate machine state as instructions and memory requests propagate.

This approach is similar to the Asim approach in that it requires the user to have confidence that the backend emulator is modeling state transitions exactly as their desired timing model intends. Just as with Asim, any deviation from the supported state transitions requires either addition or modification to the emulator. This necessarily implies that a simulator user who wants to be able to change the model topography radically by reconnecting modules will also need to have a firm understanding of the emulator implementation. Furthermore, the user may have to maintain and update the emulator. Some changes, such as adding multiple processors, may require nontrivial modification to the emulator. With regard to NIC design, the asynchronous and unsolicited memory interactions would have to interact with the emulator. Again, ensuring that these interactions happen in the order modeled by the simulator timing may require a full audit of the emulator code. In short, the emulator introduces a substantial software engineering burden that may severely limit a designer's ability to evaluate a wide scope of designs correctly.

Spinach addresses these issues by taking advantage of the LSE's intermodule dataflow support. Furthermore, using the emulator backend is not an LSE requirement, and Spinach simply does not use it. Rather than using the modules for timing only and maintaining state in the emulator, Spinach modules send and receive state data via their ports. Some modules, such as Spinach memories, also maintain local state. Memory modules receive memory addresses and load or store local data via their ports, ALUs perform requisite arithmetic operations on intermediate values, processor control signals flow throughout the model via channels, and interconnect modules, steered by the values generated by control modules, route data throughout the simulator model.

Hence, the LSE provides a straightforward method for mapping hardware structures to simulator modules. Spinach extends this mapping by coupling module behavior directly to state changes and dataflow. Since LSE port dataflow happens on a cycle-by-cycle basis and Spinach ties state to timing, Spinach modules naturally support the asynchronous transactions common in NICs. Spinach's coupling of timing and state increases the amount of computation per simulated instruction because all intra-instruction operations (including control generation) must be modeled completely. However, this model of simulation ensures cycle-by-cycle accuracy and correctly models any asynchronous or unsolicited simulation interactions. In the Spinach model, there truly is no difference between an asynchronous, unsolicited request for

service and a "normal" instruction request. They both proceed cycle-by-cycle through the system and complete just as they would in real hardware.

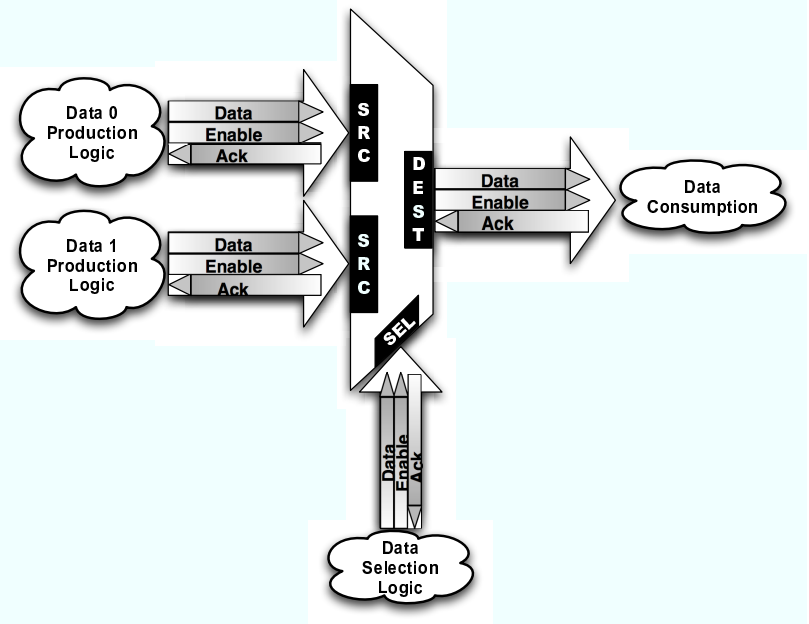


Figure 3.1 A Spinach mux with LSE channels enumerated.

The difference between the standard LSE mux and the Spinach mux underscores the distinction between the two approaches. Figure 3.1 depicts the Spinach mux. The standard LSE mux uses an algorithmic selection parameter to choose which input gets passed through. That is, there is no "select" input. The algorithmic selection logic makes a query to the backend emulator to determine, according to the current instruction, what selection should be made. However, this violates Spinach's simulation abstraction model. In Spinach modules, decisions must be local and depend only on the inputs to the module and state maintained by the local module. There-

fore, Spinach's mux module has a "select" input represented as a simple integer that determines which input gets passed through to the output. (The select input must be generated by another control module, which likely takes current-cycle state values as inputs to determine where to route data.) Unlike the simplified figure, inputs and outputs are uniform-sized vectors; the "select" signal chooses a vector of inputs to pass through.

It is important to notice that, under the Spinach simulation model, module state is completely independent of the rest of the system. Thus, adding new components (such as processors or memories) or reorganizing existing components does not affect the correctness of the rest of the system. The resultant simulator will faithfully model execution and memory behavior exactly as the user specifies by the LSS module connection listing. Because Spinach modules maintain a precise mapping of hardware to simulator modules, users must understand only the potential state modifications introduced by their added or changed "hardware" (module) connections. Thus, users have plenteous degrees of freedom with which to modify their configurations, all by simply adding and changing high-level LSS module connections. Furthermore, the user does not have to maintain a sequential emulator, which leads to lower software engineering costs. Spinach's low software engineering cost enables rapid exploration of a wide architectural design space.

3.2 Spinach Modules

The vast majority of existing LSE modules do not satisfy the Spinach model of simulation. In order to build a full Spinach-based system, one must have sufficient Spinach-compliant building blocks. This section introduces the Spinach library of base modules. This library consists of 21 new modules, which may be placed into four general categories: simple interconnect, memory, input/output (I/O), and processor. Interconnect modules act as glue logic between the various other computation and memory modules. Memory modules are those that hold memory state and control timing and arbitration of general-purpose memory systems. I/O-specific modules are those that interact with the external host and network. Processor modules are those used in a processor core and may include instruction-set specific modules for decoding.

3.2.1 Interconnect

Table 3.1 presents the Spinach interconnect modules. These modules typically function as glue logic between data-generating modules and state modules. The "inputs" and "outputs" listed in this and all subsequent tables are LSE ports. Ports connect to each other via LSE channels, each of which contains data, enable, and acknowledgement subparts. Figure 3.1 illustrates how a mux might be used as glue logic between source logic and destination logic that will consume the value(s) selected by the mux. All interconnect modules that act as steering or pass-through logic pass the contents of the entire LSE channel: data, enable, and acknowledgement (Ack).

Module Name	**Inputs**	**Outputs**
Ack Detector	Data	Passed-through Data, "Ack Detected" Signal
Data Detector	Data	Passed-through Data, "Data Detected" Signal
Demux	One *N*-wide Vector, Select	*M N*-wide Vectors (one is valid)
Function	Data	Resultant Signal
Mux	*M N*-wide Vectors, Select	One *N*-wide Vector
Pipe*	Data	Time-delayed Data
Receiver	Data	(none)
Sender	(none)	Data
Tee*	Data	Fanned-out data

Asterisks (*) denote modules distributed with the Liberty Simulation Environment
Italics denote algorithmically parameterized modules

Table 3.1 Spinach interconnect modules.

Ports that are not passed through receive a negative acknowledgement (Nack).

Both the Ack and Data detectors analyze a passed-through LSE channel for the presence of positive acknowledgement and data, respectively. Such modules are useful in control units to analyze progress of variable-latency circuits, such as memory hierarchies. The Function takes advantage of LSE's capability to provide algorithmic parameters; it takes several input data signals and applies the user-supplied algorithm to determine its runtime output. Notice that the Tee (which acts as a fanout, since LSE channels may not be multiply tapped) and Pipe (a pipelined delay unit) are the only LSE-supplied modules; all other modules are new.

3.2.2 Memory

Table 3.2 enumerates the Spinach memory modules. Users compose these modules with interconnect modules to create memory organizations. Memory modules, like all other Spinach modules, are general purpose and may be used generically throughout a system model. For example, instances of Multiported Memory modules would be used both as register files and as main memory. Memory modules commonly accept and repeat memory references. Spinach memory references contain an address, an enable that indicates a load or store request, any data to be stored, and the reference's datatype. The datatype specifies the size of the memory reference. Both data and addresses use one LSE port instance per byte of information. While modules must decode and reassemble these addresses and data, using vectors of bytes for address and data permits modeling of arbitrary address spaces and data widths that may be far beyond that which the simulator host directly supports.

The arbiter uses an algorithmically provided parameter to determine which input reference should be passed through; this effectively sets the arbitration policy. The Block Memory Transfer unit acts as a DMA engine between Spinach's cycle-accurate, variable latency memory networks.

Both the Cache and Memory Controllers model timing policies for accessing a physical memory. A physical memory is instantiated as a Multiported Memory module, whether it is a register file, cache, SRAM, or DRAM. The Multiported Memory

Module Name	**Inputs**	**Outputs**
Arbiter	Memory References, Returned Memory Data	One Selected Reference, Passed-through Memory Data
Block Memory Transfer	2 Addresses, Length, Direction, Memory Data	2 Memory References
Cache Controller	Memory Reference, Cache/Main Memory Data, Snooped Reference	Cache/Main Memory References
Memory Controller	Memory Reference, Returned Memory Data	Memory Reference, Passed-through Memory Data
Multiported Memory	Memory References	Memory Data
Scratch Pad Filter	Core Memory References, Returned Memory Data	Register/Scratch Pad/Main Memory References, Passed-through Memory Data

Italics denote algorithmically parameterized modules

Table 3.2 Spinach memory modules.

instance maintains each local memory state, but it does not model any intercycle latencies. Rather, the Multiported Memory retires P references per simulation cycle, where P is the number of input ports. Reads get processed before writes, thereby enabling intracycle data forwarding.

The Cache Controller maintains tag state for all cached blocks and uses a write-through, no-read-allocate policy. Additional coherence policies may be supported by modifying this module. The Cache Controller satisfies incoming requests by forwarding the request to a variable-latency cache unit or by loading the corresponding cache line from main memory.

The Memory Controller maintains burst access information about the recent memory patterns in order to determine when the memory bus is available. Requests that arrive during waiting cycles receive a Nack. The Memory Controller may be configured for either DRAM or SRAM operation. The Memory Controller issues memory references in order. Future improvements may include a memory access scheduling policy such as that discussed by Rixner et al. [40] in order to increase achievable bandwidth, but such policies require a complex split transaction bus. Such a bus architecture may not be feasible in embedded systems such as NICs. For DRAM operation, user-supplied row length and row activation latencies determine the latency each subsequent access experiences. SRAM operation is just a subset of DRAM operation; SRAMs have a smaller burst ("row") length and have fixed latencies between

bursts regardless of any switch between reading and writing.

A Scratch Pad Filter unit serves as a special-purpose arbiter and memory reference steering unit. Embedded systems commonly have private, low-latency scratch pad memories that may be partitioned from the global address space; the scratch pad filter detects private references and forwards them accordingly to either scratch pad memory or partitioned memory-mapped registers.

3.2.3 Input/Output

Module Name	**Inputs**	**Outputs**
DMA Assist	Read Host Data, Memory Data	Host Reference, Memory References, Write Host Data
MAC Assist	Memory Data, Receive Network Data	Memory Reference, Transmit Network Data
NIC Feeder	Host Reference, Write Host Data	Memory Reference, Read Host Data
Packet Feeder	(none)	Frame Data

Table 3.3 Spinach input/output modules.

A network interface card must interact with a host and with an external network. The modules in Table 3.3 facilitate these interactions. The direct memory access (DMA) and medium access control (MAC) Assist units reside "onboard" the NIC. The NIC and Packet Feeders are part of the overall simulator, but they model the necessary external subsystems that drive NIC operations.

DMA Assists transfer data between the NIC's host and local NIC memory. These assist modules autonomously interact with NIC memory as they constantly read pointers that indicate whether or not transactions are available for processing. If so, the assists then read descriptors which characterize the transactions to be processed. The DMA Assist negotiates arbitration with the host (a NIC Feeder instance) and transfers the data cycle by cycle. Since NICs need to read and write host memory, the DMA Assist may be configured for reading or writing. One instance implements each type of operation. Having separate instances enables concurrent processing of read and write transactions that are in different stages of processing.

The MAC Assist module is analogous to the DMA Assist. MAC Assist instances facilitate the transfer of data between NIC memory and the external network. Like the DMA Assist, the MAC Assist polls descriptor pointers that indicate when frames are ready to be sent out on the network. When frames are available, the MAC assist reads descriptors for each frame out of NIC memory and enqueues it in a local assist FIFO buffer. Frame data drains from this FIFO at the speed of the MAC. Receiving frames requires the MAC Assist to inspect arriving frame headers for the length field in order to build an appropriate descriptor. As frames complete, the MAC Assist transfers data to NIC memory, builds appropriate descriptors for the NIC processors to read, and updates the producer value indicating that more frames have arrived. As with the DMA Assists, one NIC usually has two MAC Assist instances (one each for

transmit and receive.) The MAC Assists also timestamp frame departure and arrival for offline statistics analysis.

Together, the NIC and Packet Feeders act as a harness to the simulated NIC system and are used to mimic interactions external to the NIC. The Packet Feeder sends trace data from a file into the receive side of the NIC at a rate parameterized by the user. This trace data corresponds to received UDP data. The NIC Feeder mimics the host interactions, such as writing the NIC mailboxes and servicing DMA requests, that are necessary to facilitate NIC operations. Traces of send mailbox events and receive ring preallocations, which the NIC Feeder reads from a file, determine the mailbox-writing behavior during simulation. During simulation, the feeders play back these traces as fast as possible while maintaining a balanced ratio of actions of each of the three types at every point in the simulation. In a real system, sends and receive would actually be correlated (e.g., TCP data transmissions and acknowledgements), but this behavior is not directly exploited by the Ethernet layer and is not captured by this simulation harness. A more complex harness could capture these behaviors by using TCP connection identifiers to determine which frames have been processed by the simulated NIC.

The NIC Feeder may add a parameterizable amount of fixed bus transfer overhead. Results reported by Kim et al. show that PCI bus overhead is about 30% while processing NIC workloads [25]. Other parameters include the rate at which send-

and receive-mailbox events occur and the send- and receive-coalescing factors. The coalescing factors determine how often a traced mailbox event actually gets written as a mailbox event during simulation. The NIC Feeder collects frame departure and arrival statistics just as the MAC Assists do. Having statistics from both sides of the NIC (host and network) provides insight into how long frames take to process and into the sustained rate of frame transmission and receiving during full-duplex operation.

3.2.4 Processor

Module Name	Inputs	Outputs
ALU	2 Operands, Opcode	Result
MIPS Decoder	MIPS Instruction	Data-Stationary Control Signals
Fetch Unit	PC, Stall/Squash Indicators, Memory Data	Memory Reference, One Fixed-length Instruction
Functional MIPS	Fetch/Mem Data	Fetch/Mem Memory References

Table 3.4 Spinach processor modules.

Spinach processor modules model processor-specific behavior. Like the multi-ported memory, the ALU models single-cycle access; additional timing requires external latency units, which may be combined hierarchically. The MIPS Decoder takes one 32-bit MIPS R4000 instruction as input and generates all the control signals necessary to implement that instruction as it flows throughout the processor's stages.

The Fetch Unit acts a state machine with variable delay to interact with the memory system. Given an instruction address, it fetches one 32-bit instruction.

The Functional MIPS module is a whole-processor replacement for a detailed MIPS processor core. It maintains local state and performs all calculations locally without referencing any simulator-wide global state mechanism such as the LSE emulator. Furthermore, the Functional MIPS module interacts with the system just as all other Spinach modules do. However, it does not model a processor pipeline. Each instruction fetches during one cycle and retires during the next (if the instruction does not have to stall on memory.) This is similar to the Mipsy simulator described by Gibson et al. [19]. The Functional MIPS module interacts with the cycle-accurate memory system to which it is connected, but internally it executes instructions as a sequentially programmed emulator would rather than as a detailed Spinach simulator. Typically the Functional MIPS module serves as a faster processing engine for use in system initialization routines that are not sensitive to timing and that do not affect the timing correctness of subsequent simulations. A designer may wish to use a design with Functional MIPS models during initialization and then switch to detailed models later during NIC processing. This switching is analogous to the way SimOS uses Embra, the fast binary-translation simulator, during the OS boot phase [41].

3.3 Modeling a Real NIC using Spinach

Using only the modules introduced in Section 3.2, it is possible to create an accurate NIC simulator capable of running real NIC processing firmware. Section 2.1.2 introduced the Tigon programmable NIC architecture, the only existing Gigabit programmable NIC architecture. The Tigon architecture serves as a reference against which to validate the Spinach toolset.

3.3.1 Modeling Tigon using Spinach

With the Spinach library in place, modeling Tigon simply requires assembling the various Spinach modules that represent the analogous Tigon hardware structures. Spinach's one-to-one mapping of hardware to software modules makes this process straightforward, though nontrivial. Figure 3.2 shows how Spinach modules are used to model the Tigon architecture. Notice that, except for the processor cores, all of the hardware structures depicted in Figure 2.3 map directly to basic Spinach modules. All memories are implemented as multiported memory module instances. Because memory state maps directly to memory modules, software with self-modifying code executes exactly as it would on hardware. Many simulators have difficulty correctly modeling self-modifying code because instruction memory is typically loaded at simulation start time and then executed as simulated instructions proceed. Furthermore, discrete-event simulators and the LSE backend emulator execute instructions separately from the timing of instructions. For self-modifying code to work correctly,

memory effects must take place in a cycle-by-cycle, accurate manner, and each instruction read must read this cycle-accurate memory. Spinach does precisely that simply by virtue of its simulation abstraction model. The Tigon-2 firmware uses a form of self-modifying code. It loads the NIC firmware into SRAM initially, but processors may copy commonly used portions of it to local scratch pads for low-latency instruction fetching.

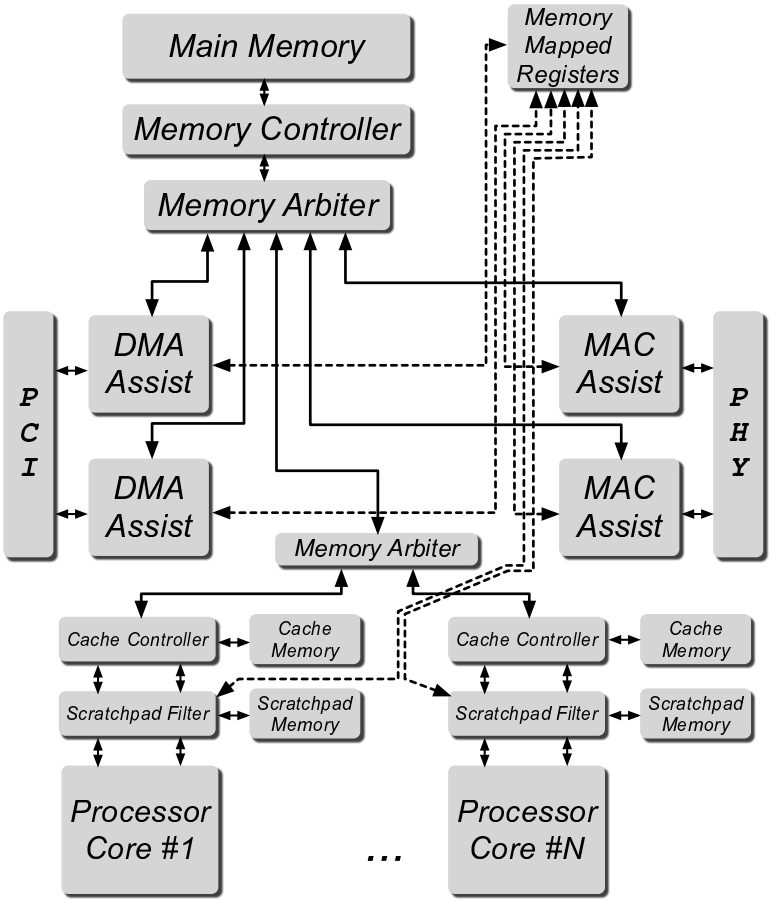


Figure 3.2 The Tigon-2 architecture modeled using Spinach modules.

Two memory arbiters are used to represent the two phases of arbitration among the processors: first, round-robin prioritization between the processors, then priority arbitration with the rest of the requesting devices. Since every module in the memory organization carries state only about whether or not the reference was acknowledged by the next higher level, stacking of multiple instances of arbiters and memories facilitates the simulation of this system's heterogeneous memory hierarchy. The processors' memory arbiter uses the base Spinach round-robin prioritization, while the main memory arbiter mimics the Tigon-2 prioritized arbitration scheme with a custom algorithmic parameter specific to that instance. The scratch pad filter modules examine processor core references and decide whether the reference should be satisfied by local scratch pad memory or by the global multiported memory-mapped registers. If the filter determines neither the scratch pad nor memory-mapped registers should satisfy the pending reference, the filter passes the reference on to higher levels of the memory hierarchy. Not shown on the other side of the boxes marked "PCI" and "PHY" are NIC Feeder and Packet Feeder modules, respectively.

Building a processor core model is easily the most complex aspect of the Tigon-2 Spinach model because of the larger number of port connections and control logic. However, once a processor core model is in place, no further maintenance is required even if the overall NIC architecture changes significantly. Processor core modules may be used just as flexibly as any other Spinach module.

Figure 3.3 depicts an implementation of a 5-stage MIPS processor core using Spinach modules. The figure bears a striking resemblance to Patterson and Hennessy's instructive logic outline of a basic MIPS core [37] because Spinach modules map directly to these architectural logic structures. Stallable pipeline registers are made out of Pipe modules for storage and Mux modules that select between the previous pipeline value and new input values. Control logic, such as the stall generator, immediate selector, or ALU input selector, is implemented as multiple function instances which use various current-instruction control signals and a user-provided algorithmic parameter in order to make selections that steer data. The memory channels are the external port interfaces to the rest of the memory organization.

Some control logic, such as the caching of branch target addresses during stall and hazard conditions, is omitted for figure simplicity. These control logic modules are hierarchical modules of functions, pipes, and muxes which connect with the rest of the core. Replacing the detailed processor models with faster functional models requires only redefining the processor core instances as Functional MIPS instances rather than as instances of hierarchical MIPS models; their port interfaces (the "memory channels" in Figure 3.3) are identical.

Thus, mapping a desired NIC architecture to Spinach modules has none of the complexities of mapping hardware structures to sequential simulator code. A designer simply connects the modules, representing hardware structures, via their ports in

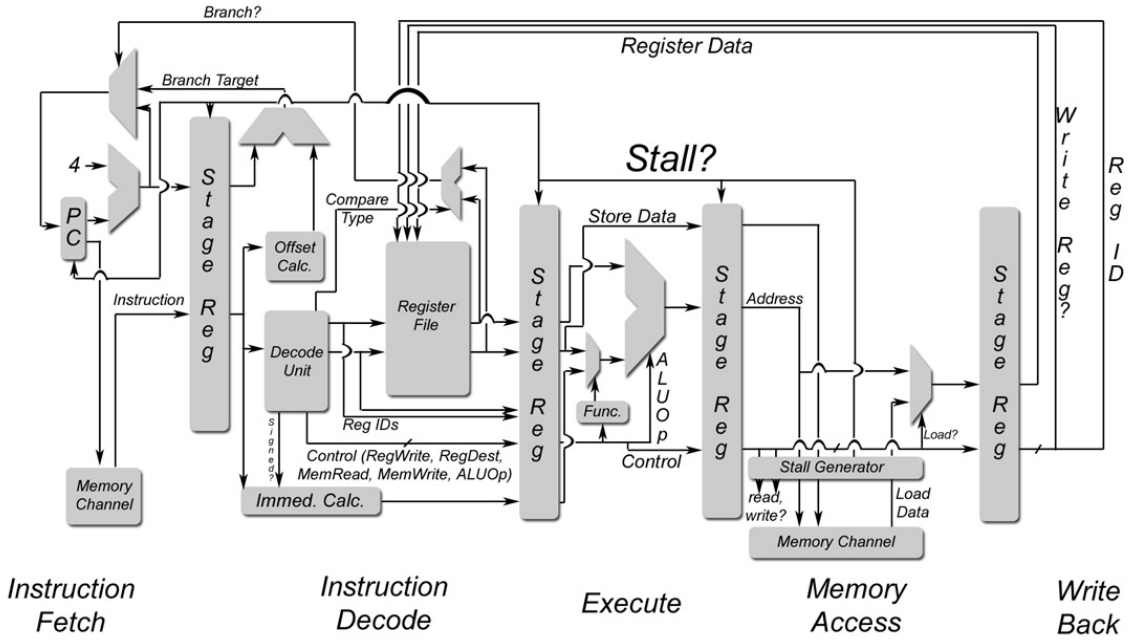


Figure 3.3 A 5-Stage MIPS core modeled using Spinach modules.

high-level LSS definition files. Notice that there may be an arbitrary number of processors connected to this architecture. To add processors, a designer must add new definitions of additional processors, scratch pad filters, and scratch pad memories, and then connect these to the processors' memory arbiter. For instance, going from a single-processor configuration to a two-processor configuration requires only 90 lines of LSS instantiation and connection code and no modification to a backend emulator. Spinach's abstraction mechanism ensures that any new state is specific to the new modules, and the new modules can interact only with existing modules via their ports.

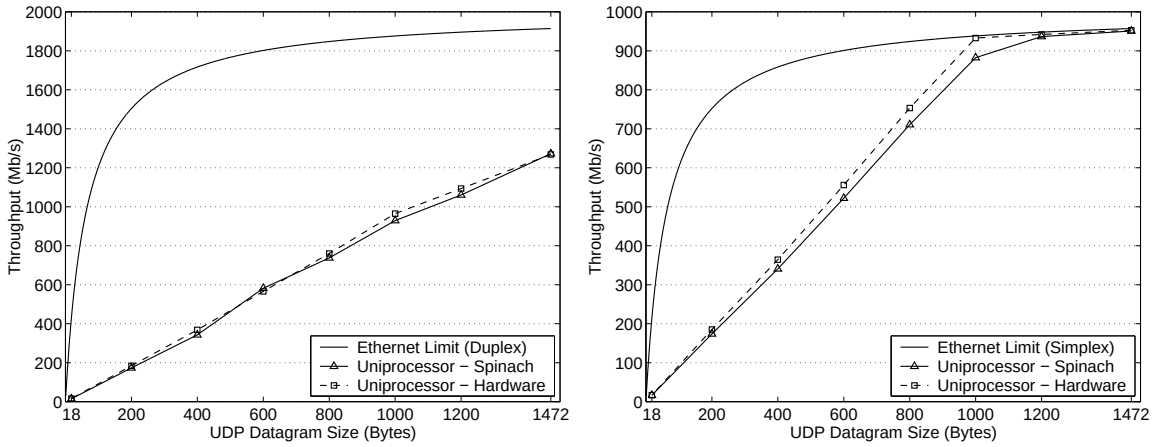
3.4 Evaluating Spinach Modeling Tigon-2

This section presents validation of the Spinach toolset by comparing benchmarks reported by a Spinach simulator that models a Tigon-2 NIC against the benchmark data collected from real Tigon-2 hardware. The transmit frame rate reported at the MAC transmit unit and the receive frame rate reported at the NIC feeder give transmit and receive rates, respectively. These combined rates give total frame throughput. When full-duplex simulations are measured, statistics are calculated using only the portion of the simulation in which frames arrived and departed. Thus, any "hang-over" unidirectional traffic at the beginning or end of simulation is discarded. This method gives a true full-duplex rate representative of what a real NIC would achieve. Unidirectional (simplex) benchmarks do not require discarding any data. Because the simulated NIC has very little state that must warm up over the course of a simulation run (e.g., no branch predictor, caches with only one line), convergence to a steady-state throughput level is achieved within as few as 100 Ethernet frames.

To analyze simulated and hardware Tigon-2 behaviors, this study uses firmware based on version 12.4.13 of Alteon's Tigon-2 firmware, which was the last version distributed [6]. Alteon's firmware uses one processor for all Ethernet data processing and uses the second processor for handling a timer loop. All versions of the Tigon-2 firmware in this paper include minor changes to speed up initialization and account for the lack of a host device driver in the simulation system, as the driver normally

initializes certain memory regions in the NIC according to the host system's configuration. The uniprocessor firmware used in this section runs all code (including the timer) on just one processor, but the multiprocessor firmware uses the parallelization strategy of Kim et al. [26]. These changes still result in valid Ethernet firmware for the actual hardware NIC, as the Ethernet frame-processing steps and code remain unchanged. When comparing hardware and simulation benchmarks, performance data from a 3Com Tigon-2 NIC provides a hardware reference.

For evaluating the benchmarks, the simulation harness (composed of NIC and Packet feeder modules) plays back the send and receive mailbox traces and the receive data trace as fast as possible without overrunning the hardware resources, as described in Section 3.2.3. Multiprocessor benchmarks attempt to drive traces at the fastest possible rate. However, various transmit-to-receive ratios are used to evaluate uniprocessor performance, and the largest measured aggregate bandwidth is reported. Typically this ratio is approximately 1:2 for both the Spinach model and the 3Com NIC. This ratio reflects the different processing requirements for the two types of frames and the prioritization of receive handlers over send handlers, a prioritization that ensures higher receive throughput on uniprocessors [6]. Regarding the send and receive processing disparity, transmit frames require at minimum two data transfers from the host (one each for the header and payload), while receive frames require only one data transfer to the host.



(a) Bidirectional Traffic (b) Receive-Only Traffic

Figure 3.4 Spinach modeling a uniprocessor Tigon-2.

Figure 3.4 shows the UDP payload data bandwidth achieved in megabits per second on a Tigon-2 NIC operating using only one of its processors. Each graph compares data from Spinach modeling the Tigon-2 (labeled "Uniprocessor - Spinach") against the performance of the 3Com Tigon-2 NIC (labeled "Uniprocessor - Hardware"). Both the simulation and hardware systems run the same version of the Tigon-2 firmware that has been modified to run on only one processor. The theoretical maximum UDP payload bandwidth on a 1 Gigabit/sec physical link is provided as a reference; receive-only traffic has a theoretical maximum representative of hits simplex nature. Notice that it is difficult to differentiate between the simulation and hardware data points, which indicates how closely the simulation models hardware behavior.

Figure 3.4(a) presents total UDP throughput for bidirectional traffic, while Figure 3.4(b) shows total UDP throughput under a receive-only workload. Receive-only workloads are used as an additional reference point for verifying Spinach's model of the Tigon-2. For all frame sizes in both the bidirectional and receive-only cases, the performance of the Spinach system is within 8.8% of the 3Com NIC's performance and within 4.9% on average. As with the 3Com NIC, the Spinach model saturates at only 66% of theoretical peak bandwidth when utilizing one processor for bidirectional traffic.

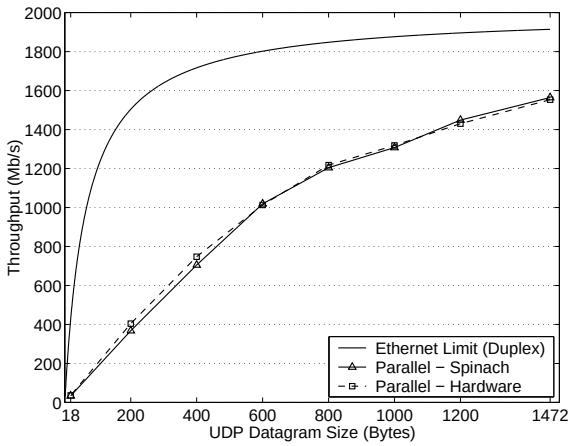


Figure 3.5 Spinach modeling Tigon-2 with parallel firmware.

Figure 3.5 presents the UDP payload throughput in megabits per second on a Tigon-2 NIC running Kim's parallelized firmware on both processors. This figure compares the 3Com Tigon-2 NIC against the Spinach model of the Tigon-2 NIC. Traffic is full-duplex. In contrast to the uniprocessor case, traffic is better balanced

on both processors since the firmware parallelization strategy puts most send-related tasks on one processor and most receive-related tasks on the other, avoiding resource contention between the processors. For all frame sizes tested, the Spinach model is within 8.9% of the hardware benchmark data and within 3.2% on average. For maximum-sized UDP frames of 1472 bytes, both the Spinach model and 3Com NIC achieve 81% of the theoretical UDP peak throughput. The parallel performance represents a 23% increase over the uniprocessor case.

3.5 Evaluating Modified Architectures

Though validating Spinach against actual hardware is a useful exercise that shows the efficacy of modeling a NIC with this toolset, the true value of any simulator arises from its ability to model systems that have not yet been implemented and to give an "inside" view to the user of simulated systems. New systems may be incremental modifications to existing architectures or radical changes to support vastly different goals. Many simulators support the former to varying degrees, but Spinach's flexibility and low software engineering maintenance support the latter. Section 3.5.1 examines using Spinach to model a Tigon-2 based system with greatly increased memory bandwidth. Section 3.5.2 presents an exploration of nonintuitive experimental results discovered in Section 3.5.1. Though this exploration requires modification of the Tigon-2's memory architecture, Spinach's flexible reconfigurability easily accommodates such studies.

3.5.1 Evaluating The Tigon-2 Bandwidth Limitation

Kim et al. hypothesized that because firmware computation requirements are invariant with frame size, limitations such as external SRAM bandwidth prevent the Tigon-2 architecture from achieving 100% of theoretical Ethernet peak throughput at maximum-sized UDP frames [26]. Furthermore, experimental results substantiate the claim that computation is not the bottleneck by showing that uniprocessor configurations can separately achieve 100% of peak throughput in one direction at a time, but that two-processor configurations cannot achieve 100% of peak throughput for both directions at the same time. Spinach permits the verification of this hypothesis since it is possible to increase the memory bandwidith while holding the processor frequency constant at 88 MHz simply by changing one parameter in the top-level LSS file.

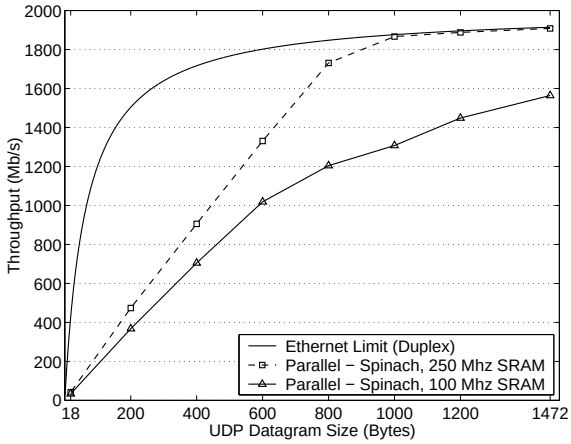


Figure 3.6 Spinach modeling Tigon-2 with 250 MHz SRAM.

Figure 3.6 compares the UDP payload bandwidth in megabits per second of the base multiprocessor configuration using 100 MHz SRAM against the same model using 250 MHz SRAM. As predicted by Kim's hypothesis, Spinach shows that the Tigon-2 architecture can fully utilize a 1 Gigabit/sec full-duplex link when given sufficient memory bandwidth.

3.5.2 Beyond Tigon-2: Improving Small-Frame Throughput

A noteworthy result not shown in Figure 3.6 is that increasing memory bandwidth also increases frame throughput for traces of minimum-sized (18-byte datagram) frames, where presumably the only bottleneck is processing capacity. Including all frame headers, the memory throughput delivered for frame contents is only 0.164 Gb/s, while the realizable peak throughput is 5.12 Gb/s. However, the frame throughput for Tigon-2 with 250 MHz SRAM is 24% more than that of the original Tigon-2 Spinach model, which serves as a baseline configuration. Spinach enables the investigation of such non-intuitive results by giving the designer the ability to modify the system in ways that may isolate the source of the performance difference.

As frame rates increase with smaller frame sizes, contention for global memory among the processors and hardware assists increases. Processors must consume and inspect more transaction (DMA and MAC) and buffer descriptors per second, and assists must also process these descriptors at a higher rate. All of these descriptors are communicated through global shared SRAM. Increasing the bandwidth to this

SRAM increases the throughput of these descriptors and decreases the average latency to access these descriptors. With the original Tigon-2 architecture, the processor cores experience higher latencies when accessing such descriptors. This problem is exacerbated by the fact that the processor cores and assists do not feature any memory latency tolerance mechanisms, such as data caching. Furthermore, processors have the lowest memory arbitration priority among all requesting units.

Further modification of the Tigon-2 architecture using Spinach emphasizes this performance issue and gives information about the tradeoffs involved. One simple modification takes the baseline configuration and uses round-robin arbitration for memory accesses. Changing the arbitration parameter to the main memory arbiter facilitates this modification. Using round-robin arbitration reduces the single-reference latency seen by the processors and assists. In an effort to reduce single-reference latencies, round-robin arbitration aggressively gives allocation to the next-arriving request and may terminate bursts. Though processor latency decreases, round-robin arbitration can reduce memory throughput significantly.

A more drastic modification is to segment the global memory space and isolate buffer and transaction descriptors from main memory. That is, all buffer and transaction descriptors are placed in a separate memory structure and accessed by a separate memory bus. The total amount of memory used by buffer and transaction descriptors is 92 KB, which could reasonably fit in one SRAM.

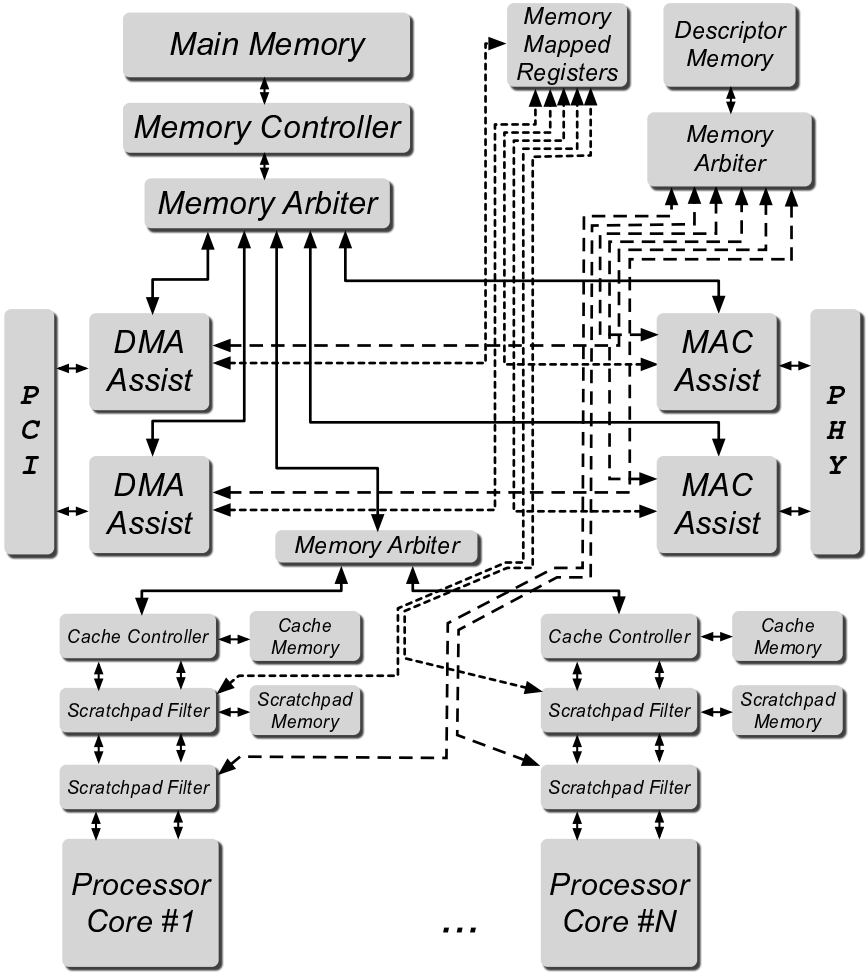
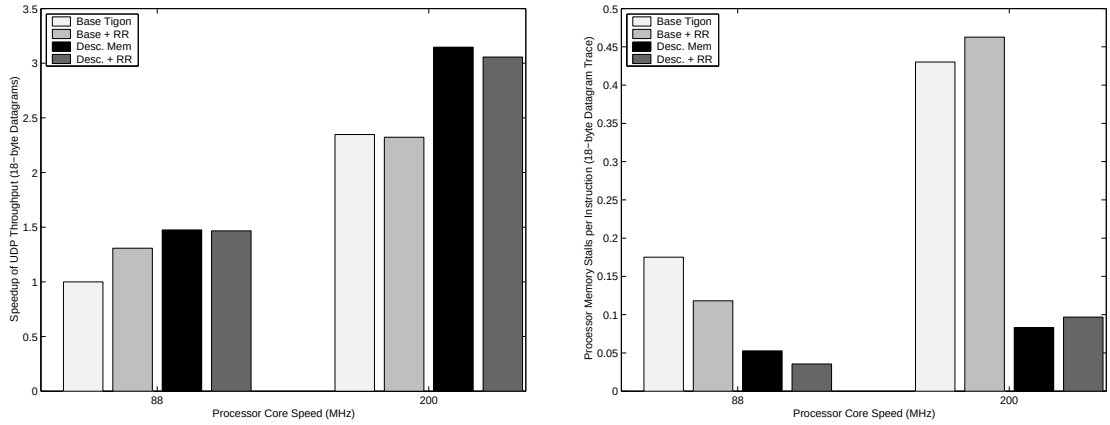


Figure 3.7 The Tigon-2 architecture with separate descriptor memory.

In order to be able to provide descriptors to all the devices that need them, both the processors and assists must have access to this descriptor memory bus. Figure 3.7 depicts an implementation of this architecture using Spinach modules. The new bus and memory are shown in the upper right-hand side of the figure. The figure shows separate connection lines for each requesting device to represent one bus with several

devices requesting arbitration. Each processor core uses another scratch pad filter module instance to check request addresses to see if requests should be satisfied on the descriptor memory bus or by the higher levels of the memory hierarchy. Each assist has a new group of ports added to it to be able to make concurrent references to the descriptor memory bus. The address range of descriptors in the global address space is user configurable.



(a) Throughput Speedup (b) Processor Memory Stalls

Figure 3.8 Evaluating architecture modifications to increase small-frame performance.

Figure 3.8 evaluates the effectiveness of various NIC architecture modifications with respect to minimum-sized frame performance. All speedups in Figure 3.8 are with respect to the base Tigon-2 architecture using 88 MHz processors. The modifications evaluated include using round-robin memory arbitration (labeled "Base +

RR"), using dedicated memory for descriptors (labeled "Desc. Mem"), and a combination of the two (labeled "Desc. + RR"). The baseline Tigon-2 architecture ("Base Tigon") is presented as a reference. In order to show the effect of increased computation resources with these new memory organizations, all configurations are evaluated with both 88 MHz and 200 MHz processor cores. For all configurations, the memory bus speed remains constant at 100 MHz.

Figure 3.8(a) shows that using the descriptor memory bus significantly improves small-frame throughput for both the 88 MHz and 200 MHz cases. Specifically, separate descriptor memories provide 47.5% and 34.0% throughput improvements, respectively. Figure 3.8(b) shows that adding the descriptor memory bus dramatically reduces the number of memory stalls per instruction, a change that provides increased processor core efficiency and increased frame-processing throughput.

Though the descriptor memory bus affects processor-memory latencies, it also decreases latencies experienced by the assist units. Thus, even though the overall instruction throughput increases by only 11.5% for the 88 MHz-"Desc. Mem" case, reduced assist latencies help improve overall frame throughput by the aforementioned 47.5%. This confirms that the base Tigon-2 configuration is not compute bound with minimum-sized frames. It confirms that assist performance with minimum-sized frames affects overall performance, since the instruction throughput increase alone does not account for the total frame throughput increase. Assists rely heavily

on descriptor throughput to complete per-transaction overheads such as fetching or storing transaction descriptors.

Though the 200 MHz-"Desc. Mem" case experiences a 25.7% increase in instruction throughput, the overall frame throughput increase of 34.0% over the base 200 MHz case is less than the increase of the 88 MHz-"Desc. Mem" configuration. As frame rates and overall accesses to descriptor memory increase, contention for descriptor memory increases and the speedup per descriptor access decreases. Thus, the per-transaction speedup contribution of the descriptor memory bus decreases with increased frame rates.

Adding round-robin arbitration does not usually increase throughput performance. The increased memory demands in the presence of decreased SRAM burst efficiency explains this relative decrease. As frame rates increase, main memory utilization increases. However, using round-robin arbitration reduces SRAM efficiency by more than 20% for all cases. This effect is particularly noticeable for the 200 MHz "Base + RR" case. Frame rate demands increase enough using 200 MHz cores that the 30% reduction in SRAM efficiency overwhelms any potential latency improvements, and round-robin arbitration decreases performance relative to the baseline 200 MHz case. In fact, the efficiency degradation combined with increased demands actually increases processor-memory latency for both 200 MHz round-robin cases, as shown in Figure 3.8(b).

3.6 Concluding Remarks

Spinach provides a useful tool to the computer architecture community that delivers the capability to model programmable network interface cards flexibly. Spinach's abstraction model couples timing to data-send operations and eliminates the notion of global simulator-wide architectural state. As a result, Spinach modules faithfully model programmable NICs, including the asynchronous memory and state interactions prevalent in them. As a verification measure, Spinach models the Tigon-2 programmable NIC architecture. The Spinach model of the Tigon-2 performs within 8.9% of real hardware performance for all cases. Spinach furthermore enables its users to modify existing NIC architectures and to create new NIC architectures. This work reveals that a modified Tigon-2-based architecture using separate memories for I/O-transaction and frame data delivers 47% better frame rates than the original Tigon-2 architecture.

Chapter 4
A Scalable and Programmable Ten Gigabit Ethernet NIC

This chapter presents a scalable Ethernet network interface card architecture for 10 Gb/s network servers. The proposed NIC architecture utilizes a partitioned, heterogeneous memory architecture that exploits the various access demands among NIC data. Power constraints motivate a parallel processing solution, and a concurrent event queue mechanism exploits the parallel resources to achieve 10 Gb/s line rates on bidirectional streams of maximum-sized Ethernet frames while it reduces core power. For instance, a simulated controller based on this architecture with six 200 MHz processor cores can achieve 99% of theoretical peak bandwidth. Using Spinach to model this multiprocessor NIC architecture yields useful information that helps optimize the firmware.

In order to sustain full-duplex line rates for maximum-sized (1518-byte) Ethernet frames, a 10 Gb/s programmable NIC must support at least 5.79 Gb/s of control data bandwidth and at least 39.5 Gb/s of frame data bandwidth. Delivering this substantial aggregate bandwidth motivates a new approach to memory organization that exploits the varying access behaviors of NIC data. Processors must access control data in order to process received frames and frames that are pending transmission. However, frame data simply must be stored temporarily in either a transmit or receive

buffer as it waits to be transferred to the Ethernet or the system host. Maximizing frame transfer speeds requires high bandwidth access to frame data, but avoiding disruptions to frame processing requires low latency access to control data. Thus, these two competing requirements motivate the separation of control data from frame data.

Extracting parallelism from NIC tasks in order to map them to many parallel, low-power computation resources requires a new event queue mechanism. The event queue controls the manner in which events are dispatched among the NIC's parallel processors, and the event queue enables frame-level parallelism. This increased parallelism enables multiple parallel cores to provide the same level of NIC throughput while using 63% less processor core power than a single-core solution.

The following section presents the computation and bandwidth requirements that define the problem space of programmable 10 Gb/s Ethernet NICs. Section 4.2 describes the hardware architecture of the proposed 10 Gigabit Ethernet controller, and Section 4.3 describes the proposed event queue firmware architecture. Section 4.4 then describes the methodology for evaluating the architecture. Finally, Section 4.5 presents and evaluates the performance of this architecture. Portions of this chapter, including some figures, are based on a technical report [49].

4.1 Supporting Ten Gigabit Ethernet

4.1.1 Satisfying Processing and Memory Requirements

A network server's host operating system uses its NIC to send and receive Ethernet frames. The OS stores and retrieves frame data directly to and from the main host memory. The NIC works in concert with the OS to transfer frames to or from the NIC's local transmit and receive buffers. The NIC and OS device driver notify each other when data is ready to be sent or has been received.

Section 2.1.1 details the steps that the OS and NIC must take to facilitate frame transmission and reception. Figure 2.1 illustrates the steps for sending a frame, while Figure 2.2 shows the steps for receiving a frame. For sending a frame, the NIC must fetch from the host the buffer descriptors regarding that frame, fetch the data pointed to by those buffer descriptors, and transmit the frame over the NIC's Ethernet interface. Receiving a frame requires that the NIC process the frame received from its Ethernet interface, build buffer descriptor(s) regarding this newly arrived frame, and transfer both the frame data and buffer descriptor(s) to the host.

A full-duplex 10 Gb/s link can deliver maximum-sized 1518-byte frames at the rate of 812,744 frames per second in each direction. A NIC that can saturate this link must sustain frame processing at this rate in both directions. Analysis of the firmware functions that implement the NIC's processing steps provides insight into the NIC's per-frame computation and memory requirements. Combining these re-

quirements with the desired frame rates yields a lower bound on computation and memory requirements that a 10 Gb/s programmable NIC must support.

Task	Instructions	Data Accesses
Fetch Send BD	24.5	19.6
Send Frame	256.9	92.0
Fetch Receive BD	12.1	12.3
Receive Frame	241.0	98.8

Table 4.1 Average number of instructions and data accesses to send and receive one Ethernet frame on a programmable Ethernet controller.

Table 4.1 summarizes the per-frame instruction and memory requirements for sending and receiving frames. These statistics are a summary of each event handler's requirements that implement the NIC's processing steps. The Fetch Send BD and Fetch Receive BD tasks read buffer descriptors from host memory that specify the locations of frames pending transmission or of preallocated receive buffers. The Send Frame and Receive Frame tasks process the DMA transactions for frame data and process the interactions with the transmit and receive Ethernet interfaces. Sending frames at peak rate thus requires instruction throughput at a rate of 229 million instructions per second (MIPS) and 2.90 Gb/s of control data bandwidth. Analogously, receiving frames at the line rate requires 206 MIPS of computation and 2.89 Gb/s of control data bandwidth. These tasks and the derived requirements depend upon only the basic task handlers and associated assist memory accesses. These requirements do not include any dispatch or synchronization overheads. Thus, the combined re-

quirements of 434 MIPS of computation and 5.79 Gb/s of data bandwidth establish a lower bound only on frame processing overheads.

While these requirements address frame processing, they do not include the bandwidth requirements of the frame data as it enters and leaves NIC memory. Each sent or received frame is first stored in local NIC memory and then later read from NIC memory after processing. For example, received frames first arrive from the Ethernet and are immediately stored in local NIC memory. After the NIC processes the received frame and prepares a DMA transaction that will write the frame to host memory, the DMA interface reads the received frame from NIC memory. Thus, the NIC must access each frame twice. As a result, sending and receiving maximum-sized frames at 10 Gb/s requires 39.5 Gb/s of data bandwidth. This is slightly less than what the overall link bandwidth would suggest (2*2*10 Gb/s) because the Ethernet interframe gap is not stored in memory.

4.1.2 Web Server Power Dissipation

Typically, modern webservers reside within large data centers. To reduce space overheads, data centers pack machines as densely as possible. However, in such a scenario power and cooling requirements lead to significant problems that can force machine density reduction and increase per-machine housing costs. Hence, power dissipation is a critical design constraint for network servers in data centers [8].

Table 4.2 classifies power dissipation in a modern webserver. This data was col-

Power Supply	Main Component	AceNIC	PRO/1000
CPU +12V	CPU	52.9 W	53.0 W
Motherboard +5V	PCI	13.9 W	4.3 W
Motherboard +3.3V	Memory	10.3 W	10.5 W

Table 4.2 Power dissipation of a network server streaming 1518 byte Ethernet frames.

lected from a system using the Intel D845GEBV2 motherboard with a 3.06 GHz Pentium 4 microprocessor and 512 MB of PC2100 SDRAM running RedHat Linux 8.0 with the 2.4.20-18.8 SMP kernel. Power consumption is measured and categorized with the methodology and equipment of Bohrer et al. [8]. The measurements in Table 4.2 were taken when the system was streaming maximum-sized (1518-byte) Ethernet frames. These measurements represent experiments with two different NICs: one with an Alteon Tigon-2-based AceNIC and another with an Intel PRO/1000 MT Desktop NIC. The AceNIC, introduced in 1997, is programmable. Section 2.1.2 describes the AceNIC's Tigon-2 architecture. The PRO/1000, introduced in 2002, is not programmable.

The table shows that the host's CPU dominates power consumption. However, the memory and PCI bus also consume significant amounts of power. Since network I/O is the only significant bus traffic in these experiments, most PCI bus power can be attributed to the NIC. The differences in power consumption between the AceNIC and the PRO/1000 can be attributed both to the AceNIC's programmability overheads and to the PRO/1000's 5 year technology advantage.

Every Watt matters in the dense environment of a data center. Therefore, next-generation 10 Gigabit NICs must constrain their power consumption to prevent them from dominating power consumption and to ensure power availability for the CPU and memory. Furthermore, power delivery and cooling constraints limit PCI devices to 25 W or less. While other motherboard designs may allow greater power dissipation, they do so at significant cost.

4.1.3 Embedded Processor Power Dissipation

Frequency (MHz)	200	400	600	800
Power (mW)	55	200	450	900

Table 4.3 Power dissipation of the Intel XScale processor at various operating frequencies [11].

Keeping the power consumption of a 10 Gigabit NIC low therefore requires careful consideration of possible design choices. General-purpose CPUs have typically employed high-frequency, wide-issue superscalar designs to achieve high performance. However, such processor designs dissipate significant amounts of power as shown in Table 4.2. Table 4.3 further illustrates the relationship between power and clock frequency by showing the power dissipation of the Intel XScale processor operating at various frequencies [11]. The XScale is a single-issue in-order processor with a seven-stage pipeline; its design avoids the excessive power dissipation of superscalar processing [11]. The XScale further constrains power dissipation by allowing dynamic

voltage and frequency variation. A wide variety of embedded devices use the XScale processor. As shown in Table 4.3, power dissipation increases roughly quadratically as frequency and voltage increase. Frequency and voltage are coupled because increased frequency requires a proportional increase in voltage for the same manufacturing process.

These power concerns force a programmable NIC to rely on techniques other than superscalar processing or high clock frequencies to provide high performance. Fortunately, the network interface tasks in Section 2.1.1 are inherently parallel. This parallelism includes parallelism across distinct tasks, as described in Section 2.1.3, and parallelism for the same task among different frames. It is far less costly in terms of power dissipation and area to exploit such coarse-grained parallelism with multiple processing cores than with one fast and complex core [33].

4.2 A Scalable 10 Gigabit NIC Architecture

A 10 Gigabit NIC must provide peak capacity in excess of the minimum requirements of 434 MIPS of computation and 45.3 Gb/s of data bandwidth. The need for additional resources stems from the difficulty of utilizing resources with 100% efficiency and from the desire to provide an extensible platform for future services. This section presents an architecture that meets the performance requirements of 10 Gb/s Ethernet in a power-efficient and cost-effective manner. Section 4.2.1 details the architectural implications of the data memory requirements, while Section 4.2.2 de-

scribes the architectural implications resulting from the computational requirements. Section 4.2.3 then details the NIC architecture that incorporates these memory and processor design strategies.

4.2.1 Data Memory System Design

A 10 Gb/s Ethernet controller's combined data memory requirements represent competing architectural requirements. Specifically, control data accesses by the processors and assist units require low latency to avoid excessive processor and assist-transaction stalls. Section 3.5.2 highlights the importance of low-latency access to transaction descriptor data (a subset of control data) when increasing frame rates. On the other hand, frame data requires capacity sufficient to store all in-flight frames and bandwidth sufficient to transfer 10 Gb/s streams simultaneously to and from the network and to and from the host. No single memory technology can provide a solution that is large, low latency, high bandwidth, power efficient, and cost effective. These considerations motivate the exploration of a memory organization consisting of heterogeneous components.

Hierarchical Organizations

Figure 4.1 illustrates how a traditional hierarchy memory organization might attempt to satisfy the various memory throughput and latency requirements for 10 Gb/s Ethernet; this organization is similar to that of the Tigon-2 architecture. For 10 Gb/s

Ethernet, the memory structure holding frame contents must be a DRAM since DRAM is the only technology with the requisite capacity and bandwidth. In the pictured memory hierarchy, all global data sharing occurs through the main memory, which in this case holds frame data and transaction descriptors. However, DRAM's high initial access latencies would heavily stall the processors and assist units whenever they access main memory to read or write transaction information. Section 3.5.2 shows that increased stalls on transaction information significantly reduce overall frame throughput. Furthermore, because the technology used in that study (SRAM) has a much smaller access latency, a 10 Gb/s design using DRAM would have even worse performance.

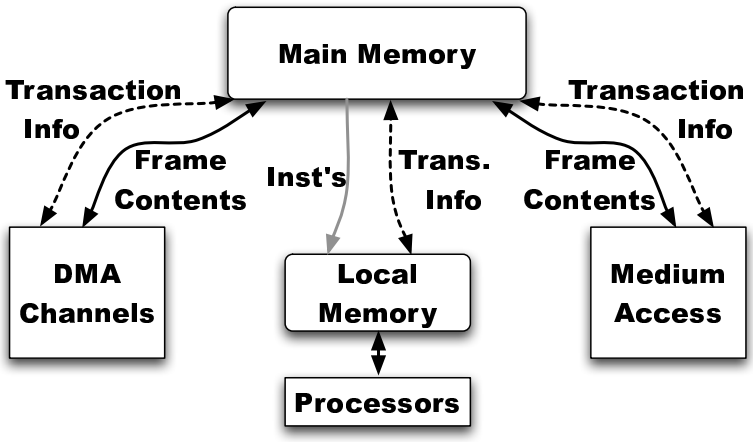


Figure 4.1 A traditional hierarchical memory organization.

The only other hierarchical organization that could satisfy the low-latency requirements of the processors and assists would be one in which the processors and

assists first access low-latency, globally coherent caches. While the coherence overheads involved may be individually smaller, such overheads are imposed per-access, and thus private references to local memory would be penalized as well. Mitigating these per-access penalties could complicate the processor core and cache controller architectures significantly and thus increase power consumption. Cost and design complexities further serve as disincentives for a cache-coherent design.

A New Approach: A Heterogeneous, Content-Specific Organization

A key observation regarding Figure 4.1 is that the competition between transaction information and frame data is artificial. Specifically, there are no data sharing patterns that require transaction information and frame contents to reside in the same memory structure. Figure 4.2 introduces a new memory organization that separates memory contents into different memory structures according to data type. With such an organization, frame contents may reside in DRAM while transaction information and other processor control data may reside in a low-latency structure, such as an SRAM. Furthermore, separating instruction memory from the other types of memory reduces contention for those other structures and reduces processor stalls due to instruction access.

Control data accessed by the NIC firmware totals about 100 KB, including transaction descriptors. This amount of data can easily fit in an on-chip memory. A single programmer-addressable scratch pad memory operating at 200 MHz with one

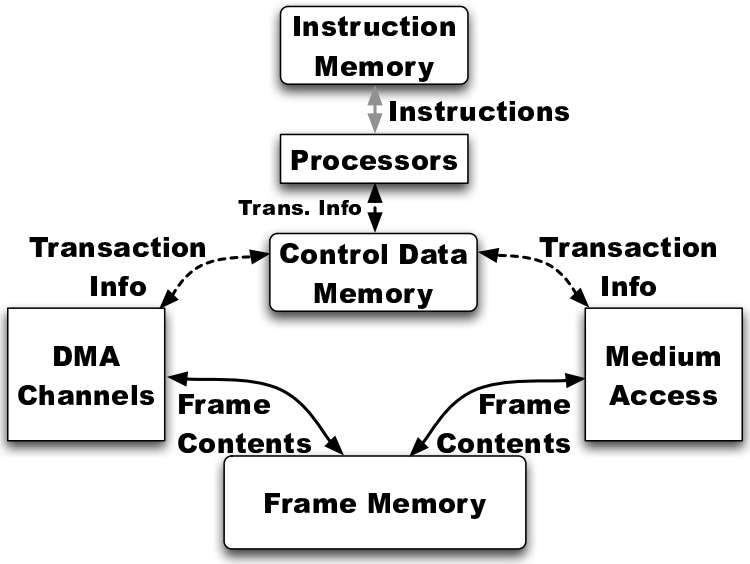


Figure 4.2 A content-specific memory organization.

32-bit port would deliver 6.4 Gb/s of data throughput, which is slightly more than the required 5.79 Gb/s. Additional memory banks are required to satisfy the additional bandwidth requirements of any dispatch and synchronization code, which the baseline 5.79 Gb/s requirement does not include. Furthermore, additional banks may be necessary to reduce latencies stemming from bank conflicts and to support future programmable functionality without reducing NIC processing throughput.

As suggested by Figures 4.1 and 4.2, processors do not need to access frame data when sending and receiving frames. Separating frame data from other memory data as depicted in Figure 4.2 frees the frame data memory solution from any artificial low-latency requirements. Notice that frame data is always accessed as four 10 Gb/s

sequential streams with each stream coming from one assist unit. Current graphics DDR SDRAM can provide sufficient bandwidth for all four of these streams. For example, the Micron MT44H8M32 can operate at speeds up to 600 Mhz, yielding a peak bandwidth per pin of 1.2 Gb/s [32]. Each of these DRAMs has 32 data pins. Hence, two of them together can provide 76.8 Gb/s peak bandwidth.

The streaming nature of the NIC's hardware assists makes it possible to achieve near peak bandwidth from such DRAM. Providing enough buffering for two maximum-sized frames in each assist ensures that data can be transferred up to 1518 bytes at a time between the assists and DRAM. These transfers are to consecutive memory locations, so using a simple DRAM arbitration scheme that allows the assists to sustain such bursts will incur very few DRAM page activations and will enable peak bandwidth throughput during these bursts.

Composing these various memory pieces provides a heterogeneous organization with specialized memories for each type of data. The resultant organization consists of banked on-chip control data scratch pads, a wide on-chip memory with small instruction caches, and off-chip DRAMs for frame data. Partitioning the memory address space according to data type provides a feasible, consistent mapping that is visible to the programmer.

4.2.2 Processor Subsystem Design

As discussed in Sections 4.1.2 and 4.1.3, the processing elements of a peripheral device such as a NIC are limited in frequency and complexity and must alternate methods, such as parallelism, to achieve high performance. Due to abundant parallelism among the tasks presented in Section 2.1.1, a high-performance network interface controller may utilize many processor cores. Furthermore, each processor core can be a simple, single-issue pipelined unit with static branch prediction. If these cores operate at 200 MHz, 3 of them would provide 600 MIPS, which is theoretically enough to satisfy the full-duplex line rate computation requirement of 434 MIPS. However, additional cores are required to account for stalls, dispatch overhead, and software inefficiencies, and to provide extensibility for future services.

Instruction fetch represents a major design consideration for programmable processors. The instructions that comprise the task-processing functions of a 10 Gb/s NIC require only about 21 KB of storage, using the Tigon-2's firmware as a basis [6]. A dedicated 64 KB instruction memory can easily fit on chip and provides the necessary storage with room for expansion. A 200 MHz instruction memory with a 128-bit port delivers a peak rate of 800 million 32-bit instructions per second. This substantially exceeds the minimum required instruction throughput. Per-processor instruction caches can be used to reduce the needed bandwidth and reduce latency caused by arbitration for the instruction memory. Working set analysis of the Tigon-

2 firmware indicates that instruction caches as small as 1 KB achieve hit rates over 95%.

4.2.3 System Integration

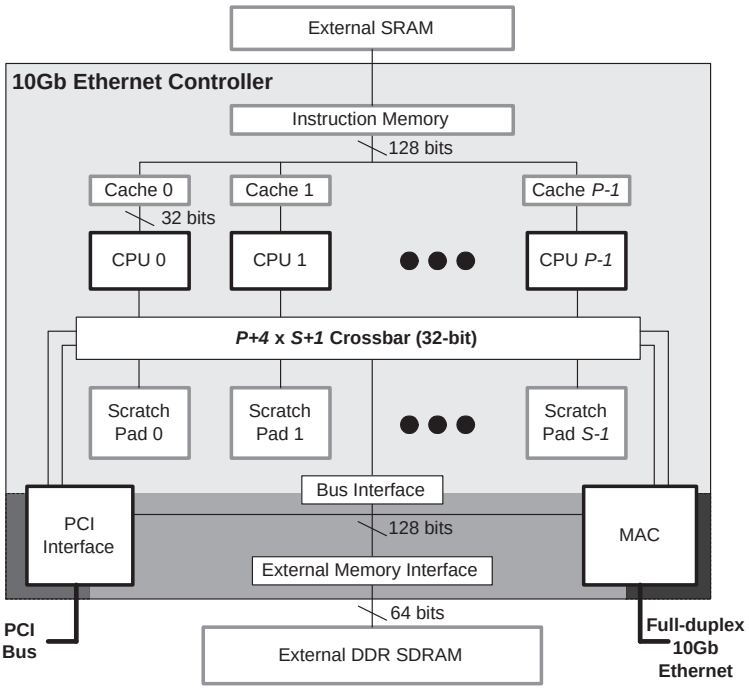


Figure 4.3 Programmable 10 Gb/s Ethernet NIC Architecture.

Figure 4.3 presents the integration of the proposed memory and computation architectures. Beyond the processor cores and partitioned memory system described above, the NIC architecture also includes hardware assist units for performing DMA transfers across the host PCI bus and for performing Ethernet sends and receives according to the MAC policy. These assist units are completely analogous to those

in the Tigon-2 architecture presented in Section 2.1.2.

As proposed above, a single instruction memory feeds per-processor instruction caches. These caches are 2-way set associative. The on-chip scratch pad has a capacity of 256 KB and is separated into S independent banks. This scratch pad is globally visible to the processors and assists, an organization that enables the necessary communication for assists to read and update transaction descriptors. The scratch pad also enables low-latency access to control data and low-latency interprocessor data sharing.

As in a dancehall architecture, the processors and each of the four hardware assists connect to the scratch pad through a crossbar. An additional crossbar connection allows the processors to connect to the external memory interface (EMI), while the assists access the EMI directly. The crossbar is 32 bits wide and allows one transaction to each scratch pad bank and to the EMI per cycle with round-robin arbitration for each resource. Accessing any scratch pad bank requires a latency of 2 cycles: one to request and traverse the crossbar and another to access the memory and return requested data. The access latency could be reduced to a single cycle and the crossbar could be eliminated if each core had its own private scratch pad, as in the Tigon-2 architecture. However, each core could then access only its local scratch pad or would require a much higher latency to access a remote location.

The processor cores and scratch pad banks operate at the CPU clock frequency,

which could reasonably be 200 MHz in an embedded system. Operating at this frequency, 3 cores and 1 scratch pad bank could satisfy the computation and control data requirements described in Section 4.1.1 if the cores operated at 100% efficiency and did not have to manage event dispatch. Providing sufficient bandwidth for bidirectional 10 Gb/s data streams requires that the external memory bus be isolated from the rest of the system since it must operate much faster than the CPUs. Forcing the CPUs to operate faster simply so that the external memory can provide sufficient frame data bandwidth is not desirable because of the extra power dissipation this would incur.

The PCI interface and MAC unit share a 128-bit bus to access the 64-bit wide external DDR SDRAM. The memory bus is twice the width of the SDRAM interface so that the bus need not operate at twice the SDRAM clock frequency to allow double data rate transfers. The bus and SDRAM may operate at 312.5 MHz to achieve 40 Gb/s of bandwidth if they can sustain 100% efficiency. However, transmit traffic introduces significant inefficiency because it requires two transfers per frame (header and data), where the header is only 42 bytes. The base architecture uses DDR SDRAM operating at 500 MHz which provides peak bandwidth of 60 Gb/s.

Since the PCI bus, the MAC interface, and the external DDR SDRAM all operate at difference clock frequencies, the NIC must have several clock domains. Figure 4.3 indicates these different clock domains by using different shadings.

The proposed architecture also provides seamless downward scalability for power savings during periods in which network traffic falls below a specified level. First, the firmware may shut off one or more processors if they are currently unneeded. Second, the decoupled clock domains allow frequency scaling of the processors and scratch pads and of the EMI depending on user needs without affecting the fixed external network and I/O interconnect interface speeds.

4.3 The Event Queue: Exploiting Frame-level Parallelism

The architecture proposed in Section 4.2 provides multiple programmable cores to meet the computation demands of 10 Gb/s NIC processing. However, the Ethernet firmware running on such an architecture must exploit sufficient concurrency to utilize the processors. Section 2.1.3 provides background on the previous NIC firmware parallelization techniques that rely on task-level parallelism and a hardware event register. Tasks are defined as steps that implement the network interface processing steps enumerated in Section 2.1.1.

While task-level parallelism via an event register provides efficient software dispatch, the primary limitation of such an approach is that there is no way to determine how many events of a specific type are pending. As a result, all events of a particular type must be processed serially. Each event handler invocation may process some or all pending events of that type. However, no other invocations of that type may proceed until the current invocation terminates, at which time the firmware can re-

examine the event register to determine if further processing is required. While this approach provides an easy mapping of the event register bit vector to outstanding tasks, the individual tasks are still too coarse grained to allow high levels of concurrency [26].

Extracting the required performance levels from the architecture in Section 4.2 requires a new parallelization approach. The event model described in Section 2.1.3 serves as a basis for firmware that supports higher concurrency levels than those provided by task-level parallel firmware. In order to achieve these higher concurrency levels, multiple frames must be able to execute the same NIC processing step at the same time. Because the event register mechanism does not permit this, this work adopts a flexible event queue mechanism. The event queue enables the *frame-level* parallelism needed to sustain full-duplex 10 Gb/s line rates on multiple lower-frequency cores.

4.3.1 Defining the Event Queue

The event queue is a circular buffer of event data structures. Each event data structure includes a type along with start and end values. The type indicates the type of event and thus the appropriate event handler. The start and end values define a range that typically represents the elements in a hardware or software buffer over which the handler should iterate. Each processor executes an identical dispatch loop that examines hardware-maintained pointers that indicate the arrival of new hardware

events. The dispatch loop also attempts to dequeue any previously enqueued events from the event queue, such as software events or hardware retries. Hardware events include completion of DMA reads and writes, frame arrival via the MAC receive assist, external BD pointer updates from the host (e.g., mailbox events), and timer interrupts. Software events include higher-level events such as "Fetch send frame contents" and "Mark receive BDs as available."

The dispatch loop checks all hardware assists and the pending software events in a round-robin fashion. This guarantees processing of at least one event of each type per loop iteration if events are available. The dispatch loop detects most hardware events when the hardware assists update their state. Hardware assists update their state by updating pointers that indicate how many hardware events they have completed. If the dispatch loop detects a completed hardware event, the dispatch loop acquires a lock protecting that specific assist's state, rechecks the state, creates an event structure with the appropriate type and range specific to the newly completed transactions, and enqueues the event into the queue, if appropriate. Synchronization is required to ensure that multiple processors do not enqueue the same event (or overlapping portions of events) into the event queue. By the time a processor acquires the appropriate lock, for instance, another processor may have already enqueued the pending events. Furthermore, there is no need to acquire the lock unless the dispatch loop determines that there may be new events.

Rather than incurring event queue insertion overhead immediately after detecting hardware events, the dispatch loop usually dispatches hardware events to their correct handlers immediately. However, if the dispatch loop determines that the proposed hardware event is large enough, it will split the event into several events. After splitting the events, the dispatch loop enqueues all but one of the events and then executes the handler for one of the newly split events.

Unlike the event register mechanism, the event queue enables event retries while still allowing other events of the same type to proceed. If an event handler is unable to consume an event fully, the handler returns a partial event that indicates the unprocessed portion to the dispatch loop. The dispatch loop then enqueues a new event corresponding to the unprocessed elements, after which other processors are free to pick up this event and try to process it again. For example, such a retry may occur as frames arrive from the Ethernet and the subsequent receive handler attempts to enqueue the frames for DMA writing to the host; if the DMA write queue is full, the firmware uses the event queue mechanism to retry rather than dropping the frames or busy-waiting.

4.3.2 Frame- vs. Task-Level Parallelism

Figure 4.4 illustrates the difference between Task- and Frame-level concurrency and introduces the manner in which the event queue enables frame-level concurrency. Both Figure 4.4(a) and (b) show how firmware processes the completion of DMA-read

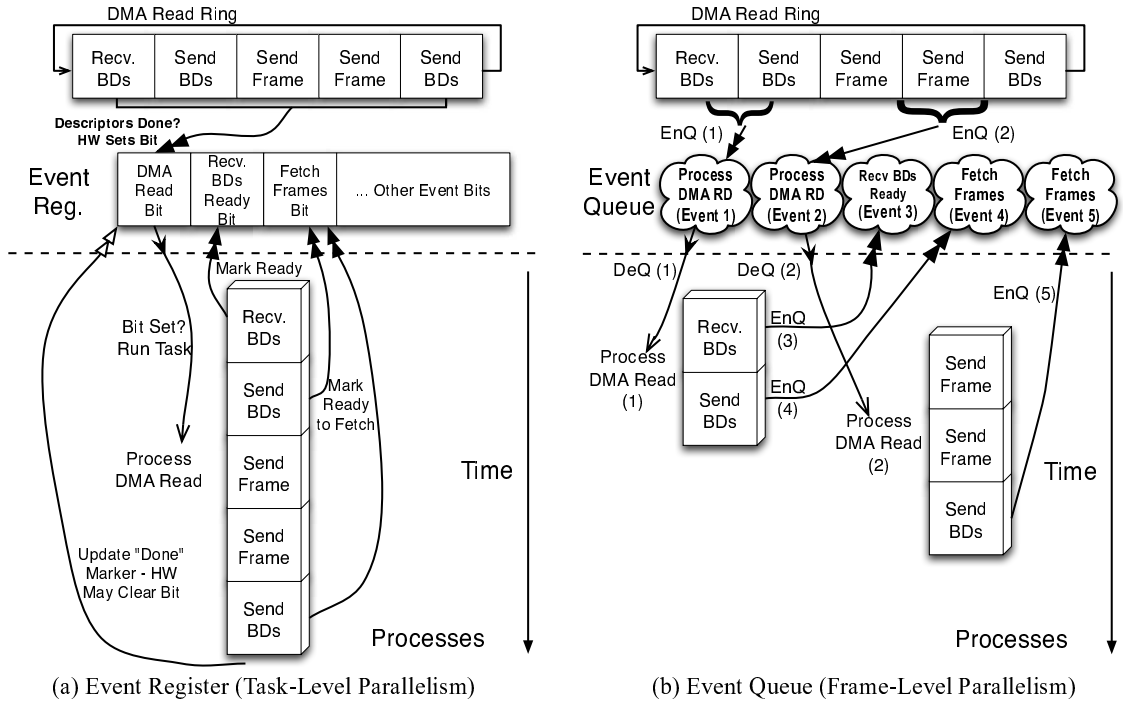


Figure 4.4 Contrasting Task- and Frame-Level Parallelism.

hardware events. In the event register case, Figure 4.4(a) shows that the hardware sets the DMA-read bit in the event register, which in turn causes the firmware to dispatch the process-DMA-read event handler. The process-DMA-read event handler inspects each completed DMA-read descriptor. Each descriptor may require further handling with future events. In Figure 4.4, the DMA-read descriptors are the boxes labeled "Recv. BDs," "Send BDs," and so forth. For instance, the process-DMA-read handler sets the Fetch Frames bit of the event register if it determines that send buffer descriptors have arrived on the DMA channel. Notice that the process-DMA-read handler must process the entire series of available DMA-read descriptors

and that while it is executing, no other instances of the process-DMA-read handler may execute. The key observation regarding frame-level parallelism, however, is that the individual pieces of such events (in this case the DMA-read descriptors that are pending processing) are independent of each other.

Figure 4.4(b) presents the same series of pending DMA-read events but shows how the event queue mechanism dispatches concurrent processing of process-DMA-read event handlers. The figure indicates event creation with filled, black double arrows and event dispatch with tailed arrows that have narrow heads. The figure associates actions with events by indicating the event number in parentheses. In this case, two processors inspect the DMA-read ring separately. The first inspection determines that two DMA-read transactions have completed, while the second finds the subsequent three transactions. Each processor immediately dispatches separate process-DMA-read handlers. Notice that unlike Figure 4.4(a), the process-DMA-read handlers may process the "Recv. BDs" transaction and the "Send Frame" transactions concurrently. Hence, the event queue enables a new form of parallelism that the previous event register construct did not allow.

4.3.3 Firmware Structure

The task processing functions in the parallelized event queue firmware are based on those used in Revision 12.4.13 of Alteon Websystems' Tigon-2 firmware [6], just as in Section 3.3. However, the code has been extended to make the task processing

functions re-entrant and to apply synchronization to all data shared between different tasks. Whenever possible, the firmware employs arrays of flags between processing steps in which only one step may produce and another step may consume. Each array element may then be checked independently for status without explicit synchronization because of the memory organization's sequential consistency model. Locks are reserved for metadata manipulation only (i.e., to check event or shared resource availability) and are released after gathering state. Event handlers do not hold locks during processing. Using synchronization sparingly enables maximum parallelization among the event handlers.

The event queue's main benefit is its ability to allow concurrent processing of in-flight frames that are in the same NIC processing step. Such concurrent events may complete out-of-order with respect to their arrival order, which implies the possibility of out-of-order frame processing completion. However, the NIC must ensure in-order frame delivery to avoid the performance degradation associated with out-of-order TCP packet delivery, such as duplicate ACKs and the fast retransmit algorithm [4]. To facilitate in-order delivery, the firmware maintains several reorder buffers where the intermediate NIC processing steps write their results. The firmware dispatch loop checks these reorder buffers in a monotonically increasing fashion, just as it checks for event arrival. If a dispatch loop encounters a frame ready to be committed, the dispatch mechanism commits all consecutive and ready frames, whether they are

intended for the host or for the network. The task that commits frames may not run concurrently with other instances of itself, but typically such a task only writes out a pointer visible to hardware so that the hardware may consume the pending receive or transmit frames. This task is designed to be as fast as possible so as to not create a bottleneck.

4.4 Evaluating a Ten Gigabit NIC Architecture with Spinach

Though the proposed 10 Gb/s architecture depicted in Figure 4.3 differs significantly from the Tigon-2 architecture, Spinach models the proposed architecture in a similarly straightforward manner. Unlike the Tigon-2 model, the proposed 10 Gb/s architecture features a crossbar that facilitates a memory network servicing concurrent references at the multiple destinations. The crossbar maps to a network of functions, multiplexors, and demultiplexors with memory arbiters at each bank. The functions select the destination bank for each memory reference; adding another memory destination that services DRAM requests facilitates the memory partitioning described in Section 4.2.1. An LSE hierarchical module composed of these function, mux, demux, arbiter, and memory connections encapsulates the entire memory system, just as a hierarchical module abstracts the Tigon-2's memory system. A separate hierarchical module within the main memory module encapsulates the DRAM controller and memory instance. From a top-level view, the two models are identical except for the number of processor instantiations and connections. Within the DRAM system hier-

archical module, the memory controller is the base memory controller that has been augmented with the row activation latencies specified for the Micron MT44H8M32 GDDR SDRAM operating at 500 MHz [32].

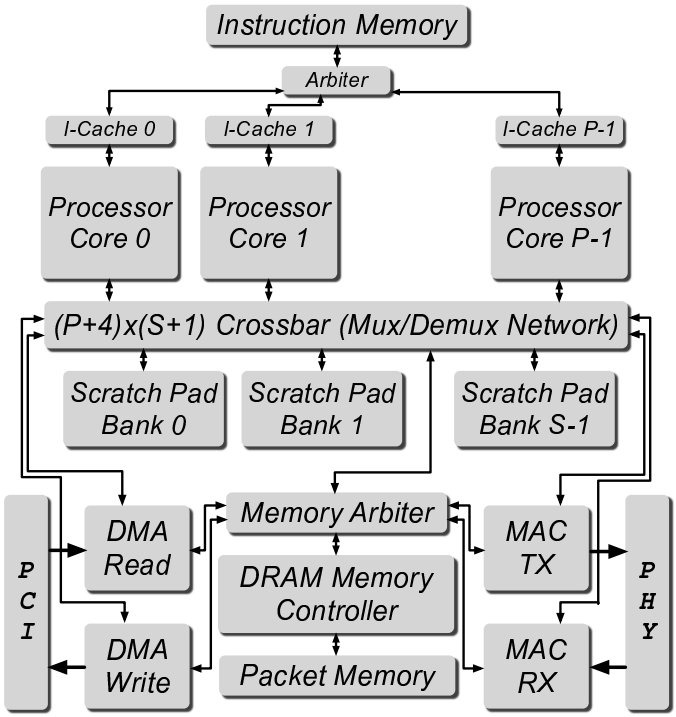


Figure 4.5 A Scalable 10 Gigabit NIC architecture implemented with Spinach modules.

Figure 4.5 illustrates the mapping of the proposed 10 Gb/s NIC architecture to Spinach modules. Note that these are predominately the same modules as in Figure 3.2. Spinach's tight coupling of module connectivity to state behavior means that new models such as this one simply require new connections, as in Figure 4.5.

Though Spinach allows flexible configuration of the various architectural pieces

of the proposed architecture, the LSE unfortunately does not directly support multiple clock domains. To facilitate the slower clocks, Spinach function instances detect whether the current simulation should be a memory or a processor cycle. This detection occurs with floating-point fidelity to allow arbitrary clock configurations. The function instances that manage clock synchronization maintain running counts of "slow" and "base" simulation cycles completed to determine whether the pending simulation cycle also counts as a "slow" cycle. If the function instances detect that the current simulation cycle should not count as a "slow" cycle, these instances inject stalls into the memory and processor systems. Because these artificial timing stalls halt all progress in every stage of the processor and memory pipelines, hazards among processing and memory stages of the entire system are maintained.

With one exception, the testing harness and mechanisms are identical to those used to evaluate the Tigon-2 model in Section 3.4. Unlike the Tigon-2 model, this study does not model host interconnect latency or bandwidth. Since server I/O interconnect standards continually evolve (from PCI to PCI-X to PCI-Express and beyond), this study focuses completely on the NIC architecture and its characteristics. As with previous evaluations, this study evaluates performance using various traces of uniform Ethernet frame sizes to determine performance at each frame size.

4.5 Experimental Results

This section investigates the performance of the scalable architecture and firmware. Section 4.5.1 discusses the base structure of the firmware and evaluates optimizations to the transmit and receive paths that increase total throughput by up to 213%. All other performance evaluations in this section use this optimized firmware. Section 4.5.2 shows the variation in architecture performance due to variation in number of cores, core frequency, and Ethernet frame size. Section 4.5.3 analyzes the architecture's computation and bandwidth utilization. Finally, Section 4.5.4 shows how varying the number of processing cores can affect power dissipation. Throughout this section, all configurations use four scratch-pad banks, an 8 KB 2-way set associative instruction cache with 32 byte lines per processor, external SDRAM operating at 500 MHz, and a physical network link operating at 10 Gb/s for both transmit and receive traffic.

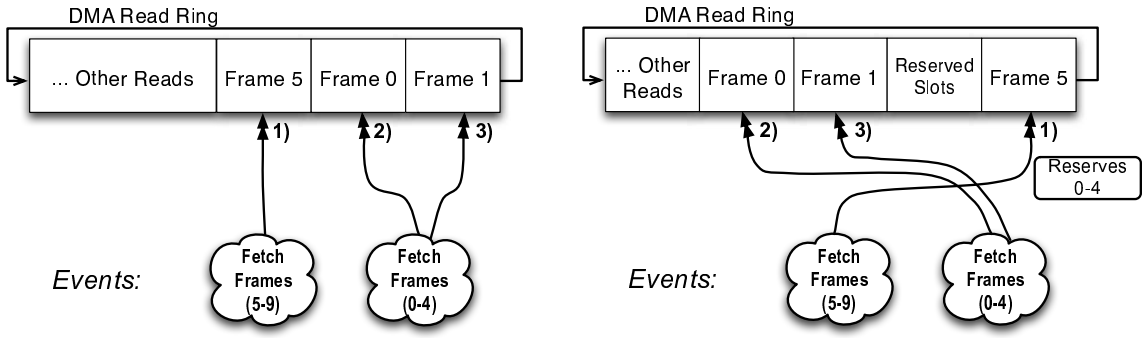
4.5.1 Firmware Optimizations

During the firmware development process, reorder buffers on the beginning and ending of NIC processing proved insufficient for all but the largest frame sizes. Specifically, throughput of send frames when processing minimum-sized frames lagged behind that of receive frames by a margin significantly larger than the margin of processing requirements in Section 4.1.1. The memory and instruction traces provided by the Spinach simulation revealed an unexpected problem: ensuring in-order frame

transmission was stalling the process of sending frames so much that the system was overwhelmingly blocked rather than sending frames.

As previously described, the event queue allows out-of-order event processing. Specifically, the events that fetch send buffer descriptors from the host may (and eventually do) complete out of order, which means that the buffer descriptors for frames later in time arrive before those which should go out earlier in time. Furthermore, the subsequent Fetch Frame events that execute after the send BDs arrive execute out-of-order, or they at least commonly overlap execution. Hence, multiple Fetch Frame events which represent consecutive groups of frames may execute concurrently. This ordering is desirable since it maximizes concurrency.

The original Tigon-2-based Fetch Frame event handler iterated over the number of buffer descriptors which represent frame portions and individually built DMA read transaction descriptors for each frame portion that should be fetched. However, the Fetch Frame event handler did not actually build the descriptor; with each handler iteration, it gave arguments to a central DMA read ring manager which enqueued those arguments as DMA requests and updated the DMA read producer. Initial firmware implementations for this 10 Gigabit architecture introduced this manager as a synchronization measure. However, relying on this manager created a situation in which the multiple Fetch Frame event handlers could make DMA requests in an interleaved fashion.



(a) Basic DMA Read Management (b) In-Order DMA Read Management

Figure 4.6 Two DMA read allocation policies.

Figure 4.6(a) illustrates this scenario. In this case, two concurrent Fetch Frame event handlers execute; the numbers beneath the DMA Read Ring indicate the order in which the DMA read requests are enqueued. Eventually frames should go out in the order in which they are indexed (i.e., Frame 0 should go out before Frame 5); committing frames to the network in order is required to avoid the application performance degradation associated with out-of-order TCP packet delivery [4]. However, the interleaved access to the central DMA request manager allowed requesting Fetch Frame handlers to request frames out of order. Since the DMA read ring is relatively small (only 32 entries by default), the ring quickly fills up. Furthermore, the simple race condition allows many "late" frames to enqueue before the "early" ones, so adding more entries does not alleviate the problem.

Eventually these DMA read requests complete, but as "late" frames arrive, transmitted frames do not get enqueued to the MAC transmit assist. Ordering constraints

require that all previous frames be completed before these "late" frames may be enqueued. The large amount of out-of-order processing that is permitted increases the likelihood that many "late" frames get enqueued in the DMA read ring, which increases the waiting time prior to MAC transmit enqueueing. The overall transmit rate degrades as simulation continues and the likelihood of out-of-order processing increases.

Figure 4.6(b) depicts a solution which enables an even higher level of concurrency while maintaining frame ordering. The same two Fetch Frame events execute concurrently, but rather than relying on a central DMA request manager for each DMA request, the Fetch Frame events request allocation in the DMA read ring directly. In this case, a central manager inspects the current ring availability and the current frame ordering in the ring. When the Fetch Frames (5-9) event requests allocation first, the request manager determines that frames (0-4) have not yet been allocated and preallocates them. It then determines where in the DMA read ring the requested frames may fit and returns a pointer to that location. Rather than writing the DMA descriptor itself, the central manager lets the requesting event do that. Careful optimization to the allocation process ensures that very little synchronization is required. When the Fetch Frame (0-4) event requests allocation, the central manager finds that Frames (0-4) have been previously allocated and updates the local state that indicates which frames have been requested.

After receiving allocated DMA descriptors, the Fetch Frame events continue and may write out the DMA read descriptors concurrently. After completing each descriptor, the events write a flag into an array that indicates what DMA read transactions are ready to be made visible to the hardware. The dispatch loop on each processor checks the status of this array in a manner analogous to hardware event detection and then updates the producer pointer. This producer update directs the DMA hardware to fetch the relevant descriptors and to complete the DMA read transactions.

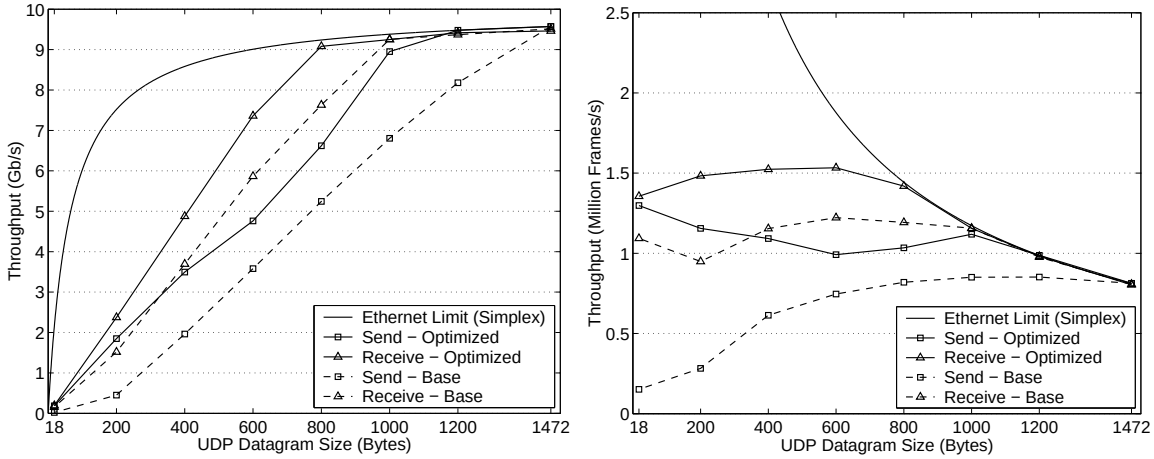
Another optimization to the base firmware was to check for available, pending transmit frames in a fashion analogous to that of checking for ready DMA read transactions. If in-order transmit frames are pending, the optimized firmware then dispatches a Send Commit event to enqueue the frames into the Ethernet transmit's interface. The base method statically scheduled an entire processor to check the status of pending frames and committed them when consecutive, in-order frames became available. While this eliminates any explicit synchronization, it wastes processing resources during periods when transmit frames are unavailable or are not arriving in order.

The optimized DMA Fetch Frame method depicted in Figure 4.6(b) enables multiple events to write into the DMA read ring at the same time, whereas the previous method required the central manager to write each descriptor serially. In order to function as a useful reorder buffer, the optimized method enlarges the DMA read ring

to 256 descriptors, increasing the firmware data footprint by 8 KB. The additional required flags further increase the memory requirement by 512 bytes. The new method ensures that DMA reads complete in-order and that DMA requests later in time cannot starve DMA requests that should proceed earlier in time. This new method for managing DMA reads results in increases in transmit performance.

Finally, the base MAC receive and DMA write processing methods each contained an extraneous lock within their event-processing loops. Replacing these locks with an array of flags requires an additional 16 KB. However, these locks blocked access to significant portions of the event loops. Again, noticing the data sharing patterns between the handlers (i.e., one produces, one consumes) permits the elimination of these locks and enables further concurrency.

Figure 4.7 shows the effects of these firmware optimizations on transmit and receive Ethernet throughput for various sizes of UDP packet streams. The NIC used in this figure has 8 processors operating at 200 MHz each. Figure 4.7(a) shows raw UDP data throughput, while Figure 4.7(b) shows throughput in millions of frames per second. For all cases, the optimized firmware outperforms the original, and the optimized firmware always experiences aggregate frame throughput increases with decreased frame size until the system saturates. Individual decreases in transmit or send traffic result from load balance inconsistencies in the firmware. These inconsistencies arise because of variations in DRAM efficiency with varied frame sizes. Furthermore,



(a) UDP Throughput

(b) Frame Throughput

Figure 4.7 Evaluating firmware optimizations for enhanced transmit throughput.

the firmware has no strict prioritization scheme; the events get dispatched according to the order processors request and achieve synchronization for each type of event as they arrive. The optimized system reaches frame processing saturation at about 2.65 million frames per second while the unoptimized firmware never exceeds 2.0 million frames per second.

Notice that for smaller frames, the transmit (and aggregate) frame throughput decreases using the unoptimized firmware as the host attempts to send frames as fast as possible. Hence, having more in-flight frames decreases frame throughput dramatically after a point. Furthermore, the optimized firmware benefits from increased available computation resources resulting from dynamically scheduling all the pro-

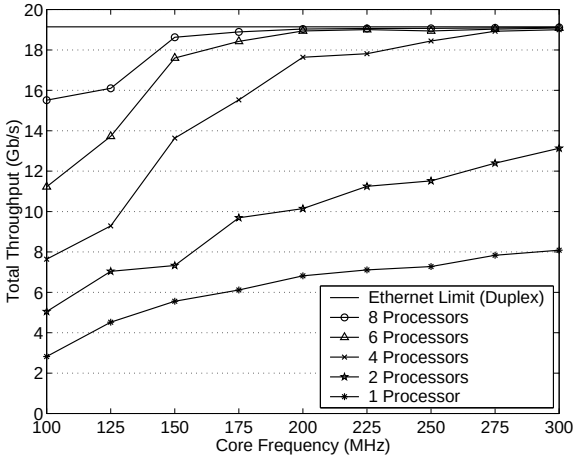


Figure 4.8 Scaling core frequency and the number of processors.

cessors to all available receive and transmit tasks. However, since at most only one additional processor becomes available as a result of dynamic scheduling, most of the performance increase on the receive path may be attributed to lock removal within receive tasks. The performance increase over the baseline firmware is most significant for minimum-sized frames; the optimized firmware's in-order DMA processing enables a 854% speedup for transmit frames. For minimum-sized frames, the combined speedup for the optimized firmware is over 213%. Furthermore, the enhancements lead to more balanced traffic, a characteristic that is key to performance in many applications [36].

4.5.2 Performance

Figure 4.8 shows the overall performance of the proposed architecture for streams of maximum-sized UDP packets (1472 bytes), which lead to maximum-sized Ether-

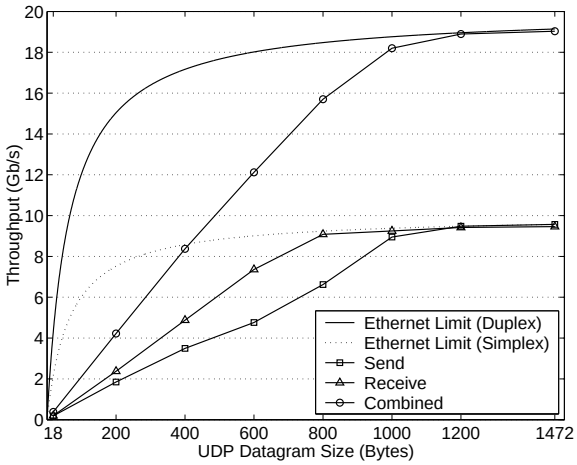


Figure 4.9 Ethernet performance, variable frame sizes.

net frames (1518 bytes). This figure shows the sustained UDP throughput as the processor frequency and the number of processors varies. Figure 4.8 shows that the NIC's architectural and software mechanisms enable parallel cores to achieve the peak network bandwidth. At 175 MHz, six processors sustain 96.3% of line rate, and eight cores sustain 98.7%. At 200 MHz, both six and eight cores achieve within 1% of line rate. In contrast, simulations show a single core must operate at 800 MHz to achieve full-duplex line rates.

Figure 4.9 presents the performance of the eight-processor architecture operating at 200 MHz across a range of UDP packet sizes. As shown in Figure 4.8, this architecture sustains 19 Gb/s, or 99.4% of the peak bidirectional throughput, with maximum-sized frames. Throughput for 800 byte to 1472 byte datagrams is close to the theoretical maximum. However, small packet performance is significantly below the theoretical peak performance. All network interface cards exhibit this behavior,

because they cannot keep up with the processing requirements of the increased theoretical peak frame rates associated with smaller frames. Most NIC processing is fixed per packet, so while bandwidth drops off only slightly as packet size decreases, the increased frame rate overwhelms the processing resources. Figure 4.7(b)'s theoretical peak bound shows the relationship between packet size and peak frame rate. While the peak rate is 812,744 frames/sec for maximum-sized packets, it grows geometrically to almost 15 Million frames/sec for minimum-sized packets. Since most network traffic consists of maximum-sized packets [3], maximum-sized frame performance is a common evaluation metric for network interface cards.

4.5.3 Computation and Bandwidth

Component	IPC Contribution
Execution	0.63
Instruction miss stalls	0.01
Load stalls	0.12
Scratch pad conflict stalls	0.04
Pipeline Stalls	0.21
Total	1.00

Table 4.4 Breakdown of computation bandwidth in instructions per cycle per core.

Six processors operating at 200 MHz very nearly saturate the bidirectional Ethernet link. Table 4.4 enumerates the ways in which the proposed architecture uses its computation resources in this scenario. Each processor has a maximum instruction rate of one instruction per cycle. During NIC processing, the processors sustain an

average of 0.63 instructions per cycle. Instruction misses, scratch pad latency, and pipeline hazards limit the computation rate. Instruction misses only account for .01 lost instructions per cycle, which confirms that the small instruction caches are extremely effective at capturing code locality even though tasks may migrate from core to core.

The six cores combined execute 141.7 million scratch pad loads per second and 97.2 million scratch pad stores per second. As introduced in Section 4.2.3, scratch pad accesses require a minimum of two cycles. If there are no conflicts, then a scratch pad load forces a single pipeline stall, while a scratch pad store causes no stalls, since the write proceeds on the second cycle without further interactions with the processor pipeline. Hence, scratch pad loads account for at least 0.12 lost instructions per cycle. Lengthening the processor's pipeline to have an additional memory stage for arbitration would alleviate these load stalls. This remains an area for future work. Fortunately, Spinach's simulation model fully encapsulates the processor units and each pipeline stage, so updating the processors is independent with respect to the rest of the system.

Scratch pad bank conflicts account for an additional 0.04 lost instructions per cycle. Techniques that tolerate the scratch pad's latency are unnecessary in an embedded system such as a NIC, but they may allow fewer processors or a lower clock frequency to achieve theoretical peak throughput. Having fewer processors or running

at a lower clock frequency have area and power benefits that may offset the increased processor complexity.

The remaining 0.21 lost instruction slots per cycle are attributed to hazards that incur pipeline stalls. This processor stalls when a load result is used by the immediately subsequent instruction, when a branch condition may not be evaluated early enough during the decode stage, and when instructions get annulled by compiler-generated branch mispredictions (as supported by the MIPS R4000). Extending the processor to support out-of-order execution would overlap load stalls with other execution, and adding a branch predictor with speculative execution could reduce both branch evaluation stalls and branch mispredictions. However, these optimizations would significantly increase the power and area of the core, which embedded systems such as a NIC cannot tolerate. These very simple pipelined cores achieve a reasonable sustained IPC and facilitate line-rate 10 Gb/s NIC processing.

Functions	**Execution Time (%)**
Dispatch	30.58
Dispatch Locking	3.28
Send	33.84
Send Locking	0.09
Receive	28.84
Receive Locking	2.48
Misc.	0.88
Misc. Locking	0.01
Total	100.00

Table 4.5 Execution Profile.

Table 4.5 profiles firmware execution time among various classifications, including Send and Receive event handlers, locking operations, dispatch loop time, and miscellaneous. This data is the result of a simulation utilizing six processors operating at 200 MHz. In the table, miscellaneous refers to metadata operations such as the DMA read manager in Section 4.5.1. Spinach's MIPS processor model introduced in Section 3.3.1 collects instruction trace information during execution that can be unobtrusively examined after execution. Combining this trace data with compiler-generated information about the firmware executable yields profile information about the simulation run.

As indicated by the event handler requirements, send processing consumes slightly more resources than receive processing. Still, the two combined only consume slightly more than 60% of the NIC's computation resources. Though the firmware's event handlers are parallelized across all six processors, the care taken to reduce in-function locking succeeds in keeping lock overheads down; they account for only 5.9% of the execution time. Unfortunately, the dispatch loop which manipulates the event queue and inspects the reorder buffers described in Section 4.5.1 proves very costly and consumes almost 31% of the execution time. While the event queue mechanism is scalable enough to enable eight lower-frequency parallel cores to sustain 10 Gb/s full-duplex line rates, implementing the event queue in software wastes a lot of resources.

The event queue's management overhead suggests the benefit of some hardware-

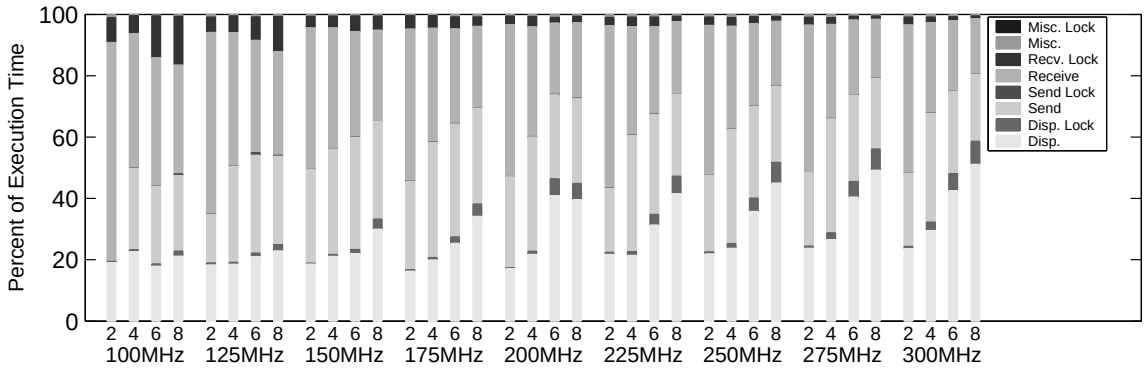


Figure 4.10 Execution profiles for varying frequencies and numbers of processors.

supported mechanism for event dispatch, such as the event register mechanism described in Section 2.1.3. Though the event register does not scale to support frame-level parallelism, its hardware management makes it very efficient. Supporting frame-level parallelism with reduced overhead would require new hardware support, most likely in which the nonprogrammable assist units are designed around an event queue mechanism and create or consume events. Even without the efficiencies afforded by such a hardware dispatch and event management system, the proposed architecture still sustains line rate for 10 Gb/s Ethernet. Scaling to higher network bandwidths increases the desirability of a hardware mechanism, however.

Figure 4.10 further enumerates how various tasks consume execution resources for various numbers of processors at each frequency between 100 and 300 MHz when processing bidirectional streams of maximum-sized packets. While the dispatch loop normally does work, it does nothing if it does not detect any events. For configurations that reach a steady-state in which both directions of traffic are roughly balanced,

dispatch time values that are more than approximately 30% represent idle time. Each event type gets checked and processed separately, so it is possible to spend less than 30% of time in the dispatch loop if the full complement of event types are not arriving. Though sending frames requires a similar amount of handler computation as receiving frames, sending requires more individual handler invocations. Each invocation requires additional event queue overhead at the dispatch loop. Hence, the relatively higher dispatch overhead associated with events makes the overall send rate lag behind when the system is heavily loaded. This event load imbalance reduces the dispatch overhead in absolute terms, and hence the shorter dispatch bars. However, it also represents a system not running at peak rate.

	Required	**Peak**	**Consumed**
Instruction Memory (Gb/s)	N/A	25.6	0.87
Scratch Pads (Gb/s)	5.79	25.6	8.28
Frame Memory (Gb/s)	39.5	76.8	39.7

Table 4.6 Bandwidth consumed by the six 200 MHz core configuration.

Table 4.6 shows the instruction, control data, and frame data bandwidth consumed when the NIC saturates its Ethernet link using six 200 MHz processors. The table also shows the minimum requirements needed to sustain line rate, as introduced in Section 4.1.1. As a reference, the table includes the peak rates provided by the six core architecture. These peak rates highlight that the architecture must be overprovisioned in order to meet line rate. For example, the 128-bit interface to the instruction

memory permits fast instruction cache fills as needed, but the instruction memory remains unused 97% of the time because of the instruction cache's effectiveness. The firmware's small code footprint enables the instruction cache's high effectiveness, but this may not be the case with other firmware that supports enhanced functionality.

Table 4.6 also specifies the data bandwidth consumed in this architecture. The processors make 238.9 million accesses to the scratch pad per second, while the hardware assists make 19.8 million accesses per second; this combines to consume 8.28 Gb/s of scratch pad bandwidth. As previously discussed, scratch pad accesses cause only 4% computation loss even though the scratch pad requires two cycles to access. As established in Section 3.5.2, the hardware assists are also sensitive to scratch pad latency since the transaction descriptors reside there. Hence, attempting to satisfy the control data bandwidth requirements with a higher-latency memory would result in an unacceptable performance loss such that the NIC could no longer sustain peak Ethernet throughput. Furthermore, the table verifies that not all the banks are needed to provide the requisite throughput, because the assists and processors consume only one third of the raw scratch pad bandwidth.

This latency sensitivity data further verifies the notion introduced in Section 3.5.2 that scratch pad latency is more important than total bandwidth for such control data. While lengthening the processor pipeline to have two memory stages (one for arbitration, another for data fetch) could reduce stalls caused by arbitration, it would

not address the issue of assist latency tolerance. Since satisfying assist latency clearly costs significant memory resources, this may be an area for optimization. While the assists prefetch as many transaction descriptors as possible, a threaded computation model similar that used by network processors to tolerate DRAM latencies [2] may help further reduce latency sensitivity. Rather than processing one transaction at a time and buffering the data of at most two transactions, the assists could process many transactions at a time and work as far ahead as possible, updating event pointers only as transactions complete. As discussed in Section 4.2.1, the higher per-access costs associated with a cache-coherent architecture and the system-wide complexity that coherent caches introduce rule them out as a feasible mechanism for reducing access latency.

Table 4.6 also shows SDRAM utilization. This SDRAM is used only for frame data. Transmitted and received frames each access the SDRAM once as a streaming store and again as a streaming load. These streams consume 39.7 Gb/s of the SDRAM bandwidth. Notice that this is slightly higher than the required throughput. This higher-than-peak usage results from misaligned accesses. Frames frequently do not arrive into the transmit and receive buffers on aligned 8-byte boundaries. When accessing these unaligned portions, the accessing unit must mask off and discard the unused bytes; these bytes represent unrecoverable, consumed SDRAM bandwidth, and hence they are included in the table. The high latency of this memory structure

(up to 27 cycles when there are SDRAM bank conflicts) does not degrade performance significantly enough to prevent the architecture from sustaining 10 Gb/s. Unlike the scratch pads, bandwidth is far more important than frame data latency. This bandwidth far exceeds the scratch pads' capabilities, further validating the partitioned memory architecture.

Though the SDRAM provides sufficient bandwidth, a memory controller that optimistically accesses DRAM in order to reduce exposed latencies could help yield similar bandwidth while using a lower-frequency DRAM [40]. The memory controller in this architecture simply blocks the entire memory bus while the current access is in progress, effectively exposing the entire row activation latency to all requesting units. Overlapping this latency would reduce the number of DRAM bus cycles occupied by such stalls. However, such an architecture requires a split-transaction bus to allow multiple in-flight requests. The overheads of a split-transaction bus and a more complex memory controller may outweigh the benefits of reduced power. This remains an area for future examination using Spinach.

4.5.4 Power versus Performance

Since the MIPS processor model used in this study is quite similar to the ARM core used in the Intel XScale, it is informative to refer to the XScale's power dissipation in Table 4.3. While the power used by this MIPS processor could be different in absolute terms, the power trend as frequency increases is likely similar because

the architectures are similar. Six processors operating at 200 MHz would dissipate roughly 330 mW, in contrast to 900 mW required by a single processor operating at 800 MHz. Hence, the event queue mechanism enables enough parallelism across the processors to yield a potential factor of three power savings, despite its dispatch overhead inefficiencies.

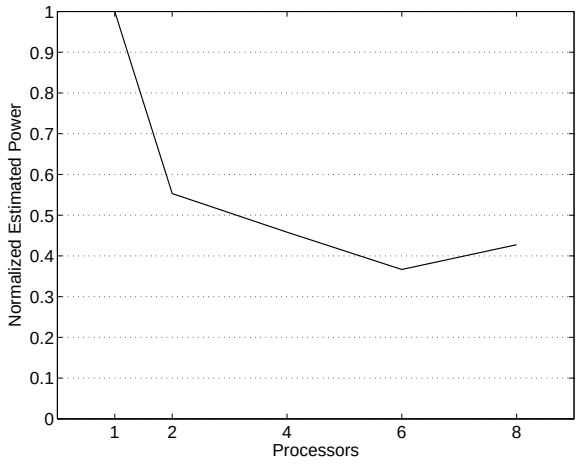


Figure 4.11 Relative minimum processor dissipation for configurations that achieve line rate.

Figure 4.11 illustrates the estimated total processor power dissipation for the lowest frequency processors that achieve line rate at each processor-count configuration. Figure 4.11 normalizes power with respect to the processor configuration that dissipates the most total processor power, which is the 800 MHz uniprocessor configuration. The power estimates result from power dissipation of the XScale processor. Since not all frequencies used in this study have corresponding XScale power dissipation figures, we use cubic spline interpolation to estimate power at unknown

higher frequencies. However, extrapolating below the XScale's minimum supported frequency (200MHz) is potentially inaccurate since the XScale's dynamic voltage variation method has a lower bound. To compensate, the 175 MHz power dissipation for the eight core configuration is linearly extrapolated to account for lowered frequency without lowered voltage.

As Figure 4.11 shows, using parallel cores to achieve the Ethernet line rate yields significant power savings. Notice, however, that the dispatch inefficiencies cause the eight-core configuration to use more total power than the six-core configuration despite the reduced per-processor power. Though the eight-core configuration is more power inefficient, a NIC that provides eight cores may be useful to firmware that supports other computationally demanding services. Additionally, an architecture featuring more cores may turn off extra processors or lower frequency during periods when the extra throughput is uneeded.

Chapter 5
Related Work

While there is no previous work defining an architectural simulator for programmable network interface cards, the field of work regarding simulated programmable systems is vast. Since even uniprocessor NICs have assist units that interact with the memory system, NICs most resemble multiprocessor systems. Several simulation infrastructures exist for modeling multiprocessors. Furthermore, while prior work does not present a NIC architecture that can fully utilize 10 Gigabit Ethernet, various other embedded architectures face related architectural issues. Section 5.1 discusses related simulation research, while Section 5.2 highlights research in embedded systems architecture.

5.1 Simulation

The Rice Parallel Processing Testbed (RPPT) simulates message-passing multiprocessors via direct execution of simulated instructions combined with discrete-event modeling of global interactions [13]. The Wisconsin Wind Tunnel (WWT) extends upon this idea by distributing the task of executing simulated instructions to multiple hosts and modeling shared-memory multiprocessors; discrete-event simulation is reserved for memory events external to the local cache [39]. Both RPPT and WWT require tight coupling between the simulation target and the simulation host, and

neither model intraprocessor detail. Covington et al. analyzed RPPT's accuracy and concluded that modeling instruction execution in detail yielded little accuracy benefit, but came at the cost of significant simulation computation overheads [12]. Such increased costs limit simulation problem sizes and can obscure details that would be exposed with larger problem sizes [15].

RPPT's processors, however, do not feature the aggressive latency tolerance mechanisms found in modern ILP processors. Pai et al. assert that the latency-tolerance mechanisms in ILP processors affect simulation accuracy even when using a highly detailed memory model [35]. To facilitate accurate study of ILP multiprocessors, Pai et al. introduce RSIM, a detailed discrete-event simulator that models ILP processors and cache-coherent nonuniform memory hierarchies [34]. Schnarr and Larus improve detailed simulation speed by detecting and caching machine state transitions in a uniprocessor simulator and then fast-forwarding state [42]. Such a technique does not map to NIC simulation since external network and bus arrivals may modify state unpredictably on a cycle-by-cycle basis.

For the network interface cards studied in this work, the in-order processors' details such as pipeline length and stall conditions affect processor throughput and hence impact frame throughput. Thus, such details are important to overall accuracy. While these are easier to consider with a less-detailed processor model than what Spinach uses, Spinach enables detailed modeling of more complex processor constructs

that can more dramatically impact accuracy (e.g., load and store buffers, out-of-order processing.) Furthermore, Spinach allows substituting a less-detailed processor model into the detailed memory hierarchy or replacing portions of the memory hierarchy with less-detailed models. This flexibility gives the user the ability to trade off detail and accuracy for simulation speed.

Gibson et al. underscore the value of validating a simulator against the hardware it models [19]. They find that detail omissions resulting from false assumptions about simulated workload behavior, such as the predicted unimportance of a translation lookaside buffer, can lead to large inaccuracies. The exercise of validating Spinach against Tigon-2 revealed similar important details, such as the effects of incorrect coalescing values on frame throughput and incorrect bus arbitration policies.

While this work evaluates NIC architectures with a harness that mimics host OS interactions, a more comprehensive evaluation would include the effects of a real host. SimOS and SimICS both model complete systems in sufficient detail to boot and run a host OS [41, 30]. Both support pluggable modules that simulate various system devices (including NICs). Combining a Spinach NIC model with a full-system simulator would require a translation layer between the system simulator's API mechanism and the Spinach model's ports. The Orion project models interconnect networks using the LSE and could be made compatible with a Spinach NIC model [47]. While combining a full-system simulator with a Spinach NIC model may give some notion of

real application performance, many other factors including the host processor and interconnect bus affect network application performance [24]. The system simulator's level of detail may or may not obscure performance details relevant to application performance.

In the space of embedded device simulation, Pees et al. use compile-time simulation to accurately simulate the TI TMS320C54x DSP [38]. Their simulation model uses a machine description combined with the workload program to analyze possible machine states. The simulator is then composed of Verilog-like processes that execute these machine states rather than directly modeling execution instruction-by-instruction. The simulator detects and runs successive states at runtime. Braun et al. [9] extend this by statically scheduling successive states at compile time. While both of these simulators model embedded processors, neither include the I/O effects present in NICs nor do they model multiprocessing. Because runtime or compile-time analysis of such effects is intractable for multiprocessor, asynchronous systems, these methods do not apply to NICs.

Of existing, published simulators, the Intel IXP simulator is most like Spinach in its functionality. Both Spinach and the IXP simulator model programmable systems and asynchronous network arrival and departure operations [7]. However, the IXP simulator is a Verilog-like tool that requires complex, low-level descriptions of module behaviors. As with Verilog simulators, significant reorganizations of the IXP module

set and extensions of existing IXP module behaviors require a significant software engineering effort.

Verilog-based simulators have an abstraction model in which user-specified descriptions model specific parts of an overall system. These parts are analogous to Spinach modules, because both Verilog descriptions and Spinach modules maintain state locally and generate output based upon user-defined inputs [27]. Unlike Verilog, Spinach modules do not internally model circuit-level behaviors, which gives Spinach a speed advantage at the potential cost of circuit-level accuracy. Like all LSE modules, the base Spinach modules are implemented in C. This higher-level abstraction makes Spinach more accessible to users who wish to extend it by providing new modules, such as an instruction decoder that supports a different ISA. Extending Verilog-based simulators in an analogous way would be significantly more complex because users typically describe Verilog modules in a complex, parallel description language. In addition to this software engineering concern, Spinach utilizes the LSS instantiation and connection language, which enables straightforward module connections and hierarchical definitions. Verilog constructs for managing instantiation and hierarchical definition tend to be more cumbersome.

5.2 Embedded Systems Architecture

Most recent embedded systems architecture research relates to network processors. Network processors (NPs) are used in routers to inspect packets and determine their

destinations according to the current routing table. Crowley et al. examine several parallelized general-purpose processor architectures via simulation and find symmetric multithreaded (SMT) architectures to be most suitable because they exploit both instruction-level and thread-level parallelism [14]. They focus almost exclusively on processing limitations and use a cycle-accurate SMT simulator. However, they only add a DMA interface to this simulator. Unlike Spinach, they model neither main memory bandwidth nor cycle-by-cycle frame packet arrival and departure.

The Intel IXP and Vitesse NPs both utilize multithreaded processor architectures [2, 46]. Like the partitioned memory in the 10 Gb/s NIC architecture presented in Section 4.2.1, the IXP2400 uses low-latency SRAM scratch pads for sharing data between processors and for managing queues between the I/O units and processors [2]. However, the heterogeneous memory organization in Section 4.2.1 results largely from the conclusions obtained in Section 3.5.2 that indicate low-latency access to descriptor and control data significantly impact both processor and assist throughput. NPs most likely rely

While NP architectures exploit thread-level parallelism found among packet-processing threads in NP applications, they also reduce the DRAM latency exposed to the processor. Hasan et al. further reduce exposed DRAM latencies and increase overall throughput by modifying the Intel IXP NP architecture to optimistically exploit DRAM row locality [20]. Such an approach is not directly applicable to the DRAM

system in a NIC because the access patterns experienced by the DRAM in a NIC are long, sequential streams. Therefore, initial DRAM latency is not a limiting factor for NICs as it is for NPs, which interleave concurrent memory accesses among router ports. However, future programmable NIC applications may require access to external DRAM by the NIC processors, which would perturb the streaming nature of the DRAM accesses. In such a situation, the proposed NIC architecture may benefit from DRAM scheduling optimizations.

While NPs typically execute one thread per packet processed, techniques such as multithreading and memory access scheduling sufficiently reduce the latencies experienced by NPs. Unlike NPs, NICs must tolerate much longer latencies, such as those associated with host interactions. Hence, NICs cannot rely on the same mechanisms to hide latency. Instead, NIC processing decomposes into events that may queue for long periods of time. Though the computation model varies significantly between NPs and NICs, some of the memory bandwidth and latency concerns are analogous.

Though NP architectures are similar to those of a programmable NIC, they do not relate to improving server performance. In support of 10 Gb/s servers, Hoskote et al. built a prototype programmable TCP offload accelerator board [21]. Unlike the proposed 10 Gb/s NIC architecture in this thesis, their offload accelerator board processes packet headers only and thus does not propose or evaluate a solution to delivering high-bandwidth frame performance while satisfying low-latency processing

requirements. Furthermore, they use a single 5 GHz processor that dissipates 6.39 W, whereas the processing architecture introduced in Section 4.2.2 utilizes several low-power, low-frequency processing cores. Though the TCP accelerator developed by Hoskote et al. leaves little available power for additional NIC processing and frame accesses, the accelerator serves as motivation for future 10 Gb/s offload architectures. Such future accelerators could be integrated into a programmable 10 Gb/s NIC architecture, such the one introduced in this thesis. As highlighted by the power trends presented in Section 4.5.4, future accelerators may benefit from a parallel approach that can reduce processor power consumption.

Chapter 6
Conclusions

As network link speeds and user demands continue to increase, computer architects will continue to find solutions that maximize Internet server performance. Realizing the full potential delivered by next-generation network links will require a network interface card that can process full-duplex 10 Gb/s streams of Ethernet frames. In response to that need, this thesis contributes a programmable 10 Gb/s NIC architecture that can saturate a full-duplex 10 Gb/s link. Programmability offers extensibility to offer network services at the NIC and removes processing burdens from the server's host processor, which enables the server to fully utilize the provided link bandwidth.

Since NICs must adhere to the tight power constraints of server peripherals, the design proposed in this thesis focuses on achieving high performance through simple, power-efficient parallel cores. Utilizing these parallel resources requires a scalable system for managing NIC processing that enables concurrent execution of Ethernet frame-processing steps. Unlike previous task-level parallel approaches, this thesis contributes a frame-level parallel firmware based on an event queue mechanism that permits multiple Ethernet frames to execute the same frame-processing step at the same time. While a naive frame-level parallel firmware functions correctly, firmware analysis shows that fetching frames out of order leads to performance degradation for

transmitted frames since the firmware must commit frames in order to the network. Making use of this analysis to enhance the firmware yields an 854% improvement in transmit frame throughput and a 213% improvement in overall frame throughput for minimum-sized frames, while still maintaining correct ordering. The resultant enhanced firmware enables a NIC with six simple processor cores operating at 200 MHz to saturate a full-duplex 10 Gb/s Ethernet link when processing maximum-sized Ethernet frames. Though the event queue mechanism introduces about 30% inefficiency in NIC processing, the event queue still enables enough parallelism such that a six-core configuration uses approximately 60% less core power than an equivalently performing uniprocessor uses.

While parallelism provides a solution to computation requirements, satisfying the 45.3 Gb/s of data bandwidth required to process full-duplex 10 Gb/s Ethernet streams also motivates a new approach to memory organization. Specifically, this work proposes a partitioned, heterogeneous memory organization that exploits the different memory access requirements among the different data in a 10 Gb/s programmable NIC. Rather than attempting to satisfy the varying memory demands with a traditional memory hierarchy, the proposed memory organization uses GDDR DRAM operating at 500 MHz to satisfy the high-bandwidth needs of frame contents and uses a 4-way banked scratch pad to satisfy the low-latency demands of control data.

Designing complex hardware and software systems, such as the proposed 10 Gb/s

NIC architecture and its firmware, requires a flexible evaluation method that provides a sufficient level of detail such that computer architects can iteratively optimize and improve designs. The high costs of hardware implementations, coupled with the inherent limitations of observable hardware behaviors, motivate a simulation approach to hardware and software design. However, simulating NICs presents a unique challenge because NICs experience unsolicited, asynchronous I/O activity, but traditional systems targeted for simulation do not. Spinach is a new cycle-accurate simulation toolset that meets the challenges of asynchronous NIC behaviors and enables simulation of programmable NICs, including their processors, memory, and I/O interactions.

Though Spinach is based on the Liberty Simulation Environment, it uses a unique approach to hardware abstraction that removes reliance on a backend sequentially coded emulator to maintain machine state. Instead, Spinach modules keep their state internally. Hence, this tight coupling of machine state to timing enables a one-to-one mapping of architectural hardware structures to Spinach modules. Therefore, creating accurate, flexible simulators that support the asynchronous, unsolicited I/O actions common in NICs and multiprocessors is straightforward with Spinach. Modeling the proposed 10 Gb/s NIC architecture requires simply instantiating and connecting the relevant Spinach architectural components using the high-level Liberty Simulation Specification language. Beyond simple modeling and performance evaluation, Spinach additionally provides a useful tool to develop and enhance experimental software on

nonexistent platforms since it can give detailed information not observable on real hardware.

To validate Spinach, this work models the Tigon-2 Gigabit programmable NIC and compares the Spinach benchmarks against those produced by Tigon-2 hardware. The Spinach Tigon-2 model performs within 8.9% of the hardware for each case tested and within 4.5% on average. Using this model, Spinach provides valuable insights into architectural performance that are otherwise unobservable. For example, though the Tigon-2 is computationally underpowered when processing minimum-sized Ethernet frames, the architecture is partially bounded by memory latency experienced by processor and I/O requests for transaction descriptors. A modified Spinach model shows that offloading these descriptors onto a separate memory bus yields a 47.5% improvement in frame throughput even when processor performance remains unchanged. The importance of maintaining low-latency access to descriptor data in order to increase frame throughput further corroborates the partitioned, heterogeneous memory organization of the proposed 10 Gb/s NIC architecture.

Spinach is not limited to its existing NIC modules. While its included I/O modules are NIC-specific, the Spinach simulation model encapsulates state and I/O interactions completely within its modules. Thus, extending its functionality only requires adding any special-purpose modules and reconfiguring the existing modules as specified by a new architecture. For example, the Spinach processor model and I/O mod-

ules are completely pluggable and replaceable. Initial research by others in the Rice Computer Architecture group shows that replacing the R4000 processor core with a simple VLIW could yield a significant improvement in NIC instruction throughput. Adding new wireless transceiver modules along with a DSP-style processor model composed of existing Spinach modules would enable Spinach to model wireless architectures.

In addition to modeling other embedded architectures, the work presented here suggests improvements to the proposed 10 Gb/s NIC architecture. As shown in this thesis, the software event dispatch mechanism used in the event queue firmware introduces a reduction in efficiency of about 30%. Future work may focus on a simulated hardware-managed event queue mechanism that offers the same flexibility of the software event queue, but with less overhead. Reducing overhead would offer the ability to further reduce power consumption. Also, adding a split-transaction DRAM bus and a latency-overlapping memory controller may reduce the frequency at which the DRAM must operate, thus further reducing power consumption. Additionally, further studies may examine the benefits of using a longer processing pipeline that includes a stage for negotiating the proposed architecture's arbitration delay. Most noteably, however, this work evaluates only the efficacy of frame processing by the 10 Gb/s NIC; future work may focus on implementing additional services using the proposed NIC and evaluating its suitability as a platform for such services. As discussed in this

thesis, such services are vital to improving server performance.

Though this work evaluates NIC throughput for various hardware and software designs, the ultimate goal of high-performance NIC design is to maximize real-world network application performance. The Liberty-based Orion interconnect project could integrate with a Spinach model and replace the uniform packet streams used in this work with variable, reactive network loads. Furthermore, methods for improving server performance are taking increasing advantage of NIC programmability. In order to evaluate application performance, future work could focus on a test harness that more closely mimics a real server host and its interconnect, while the existing NIC models presented in this work offer a platform to develop and evaluate NIC offload software. A new harness that provides high-level OS calls rather than low-level packet traces could be used to evaluate the effectiveness of computation offload from the OS to a simulated programmable NIC. To run complete applications and fully evaluate performance, future work may integrate Spinach into a full-system simulator, such as SimICS or SimOS. As computer architects strive to harness the potential offered by programmable NICs, the field of work related to programmable NIC architecture and software design will continue to grow.

References

1. 3com Corporation. Data sheet: 3Com Fast Ethernet Secure Copper and Fiber Network Interface Cards. `http://www.3com.com/other/pdfs/products/en_US/400833.pdf`, 2003. Accessed April, 2004.

2. Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6–18, August 2002.

3. Mark Allman. A Web Server's View of the Transport Layer. *Computer Communications Review*, 30(5):10–20, October 2000.

4. Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control. IETF RFC 2581, April 1999.

5. Alteon Networks. *Tigon/PCI Ethernet Controller*, August 1997. Revision 1.04.

6. Alteon WebSystems. *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition*, July 1999. Revision 12.4.13.

7. Ram Bhamidipati, Ahmad Zaidi, Siva Makineni, Kah K. Low, Robert Chen, Kin-Yip Liu, and Jack Dahlgren. Challenges and Methodologies for Implementing High-Performance Network Processors. *Intel Technology Journal*, 6(3):83–92, August 2002.

8. Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The Case for Power Management in Web Servers. In Robert Graybill and Rami Melhem, editors, *Power Aware Computing*, pages 261–289. Kluwer Academic Publishers, 2002.

9. Gunnar Braun, Andreas Hoffmann, Achim Nohl, and Heinrich Meyr. Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description. In *Proceedings of the 14th International Symposium on Systems Synthesis*, pages 57–62. ACM Press, 2001.

10. Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

11. Lawrence T. Clark, Eric J. Hoffman, Jay Miller, Manish Biyani, Yuyun Liao, Stephen Strazdus, Michael Morrow, Kimberly E. Velarde, and Mark A. Yarch.

An Embedded 32-b Microprocessor Core for Low-Power and High-Performance Applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, November 2001.

12. Richard C. Covington, Sandhya Dwarkadas, J. Robert Jump, J. Bart Sinclair, and Sridhar Madala. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

13. Richard C. Covington, Sridhar Madala, V. Mehta, J. Robert Jump, and J. Bart Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11. ACM Press, 1988.

14. Patrick Crowley, Marc E. Fluczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 14th International Conference on Supercomputing*, pages 54–65. ACM Press, 2000.

15. David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.

16. Peter Druschel and Larry L. Peterson. Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202. ACM Press, 1993.

17. Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukerjee, Haris Patil, Steven Wallace, Nathan Binkert, Roger Espase, and Toni Juan. Asim: A Performance Model Framework. *IEEE Computer*, 35(2):66–76, February 2002.

18. Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.

19. Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58. ACM Press, 2000.

20. Jahangir Hasan, Satish Chandra, and T. N. Vijaykumar. Efficient Use of Memory Bandwidth to Improve Network Processor Throughput. In *Proceedings of the*

30th Annual International Symposium on Computer Architecture, pages 300–313. ACM Press, 2003.

21. Yatin Hoskote, Bradley A. Bloechel, Gregory E. Dermer, Vasantha Erraguntla, David Finan, Jason Howard, Dan Klowden, Siva G. Narendra, Greg Ruhl, James W. Tschanz, Sriram Vangal, Venkat Veeramachaneni, Howard Wilson, Jianping Xu, and Nitin Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, November 2003.

22. Intel Corporation. Data sheet: Intel PRO/1000 T IP Storage Adapter, Bridging Gigabit Ethernet to Network Storage. `http://www.intel.com/network/connectivity/resources/doc_library/data_sheets/pro1000_T_IP_SA.pdf`, 2003. Accessed April, 2004.

23. Krishna Kant. TCP Offload Performance for Front-end Servers. In *Proceedings of the IEEE Global Telecommunications Conference*, pages 3242–3247, December 2003.

24. Hyong-youb Kim. Improving Networking Server Performance with Programmable Network Interfaces. Master's thesis, Rice University, Houston, Texas, April 2003.

25. Hyong-youb Kim, Vijay S. Pai, and Scott Rixner. Improving Web Server Throughput with Network Interface Data Caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250. ACM Press, October 2002.

26. Hyong-youb Kim, Vijay S. Pai, and Scott Rixner. Exploiting Task-Level Concurrency in a Programmable Network Interface. In *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 61–72. ACM Press, June 2003.

27. Polen Kission, Hong Ding, and Ahmed A. Jerraya. VHDL-based Design Methodology for Hierarchy and Component Reuse. In *Proceedings of the European Design Automation Conference*, pages 470–475, 1995.

28. Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software Support for Outboard Buffering and Checksumming. In *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98. ACM Press, August 1995.

29. Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy. On the Validity of Trace-Drive Simulation for Multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 244–253, May 1991.

30. Peter S. Magnusson, Fredrik Larsson, Andreas Moestedt, Bengt Werner, Fredrik Dahlgren, Magnus Karlsson, Fredrik Lundholm, Jim Nilsson, Per Stenström, and Håkan Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 119–130.

31. Keith Z. Meth and Julian Satran. Design of the iSCSI Protocol. In *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies*, pages 116–122, April 2003.

32. Micron. Data sheet: 256Mb: x32 GDDR3 SDRAM MT44H8M32, June 2003.

33. Trevor Mudge. Power: A First-Class Architectural Design Constraint. *Computer*, 34(4):52–58, April 2001.

34. Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1997.

35. Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *International Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, TX, February 1997.

36. Vijay S. Pai, Scott Rixner, and Hyong-youb Kim. Isolating the Performance Impacts of Network Interface Cards through Microbenchmarks. In *Proceedings of the 2004 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, July 2004.

37. David A. Patterson and John L. Hennessy. *Computer Organization and Design (2nd ed.): The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 1998.

38. Stefan Pees, Vojin Zivojnovic, Andreas Ropers, and Heinrich Meyr. Fast Simulation of the TI TMS320C54x DSP. In *Proceedings of the International Conference on Signal Processing Application and Technology*, pages 995–999, 1997.

39. Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of

Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60. ACM Press, 1993.

40. Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138. ACM Press, 2000.

41. Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

42. Eric Schnarr and James R. Larus. Fast Out-of-Order Processor Simulation Using Memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294. ACM Press, 1998.

43. Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 57–57. ACM Press, 2001.

44. Manish Vachharajani, Neil Vachharajani, and David I. August. The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

45. Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, November 2002.

46. Vitesse Semiconductors. IQ2200 Network Processor Family. `http://www.vitesse.com/products/briefs/IQ2200_Family.pdf`, 2002. Accessed April, 2004.

47. Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 294–305, November 2002.

48. Paul Willmann, Michael Brogioli, and Vijay S. Pai. Spinach: A Liberty-based Simulator for Programmable Network Interface Architectures. In *Proceedings of*

the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems, July 2004.

49. Paul Willmann, Hyong-youb Kim, Vijay S. Pai, and Scott Rixner. A Scalable and Programmable 10 Gigabit Ethernet Network Interface Card. Technical Report TREE0402, Rice University, Department of Electrical and Computer Engineering, March 2004.