Parallelization Strategies for Network Interface Firmware

Michael Brogioli Paul Willmann Scott Rixner
Rice University
Houston, TX
{brogioli,willmann,rixner}@rice.edu

Abstract

Typical data-intensive embedded applications have large amounts of instruction-level parallelism that is often exploited with wide-issue VLIW processors. In contrast, event-driven embedded applications are believed to have very little instruction-level parallelism, so these applications often utilize much simpler processor cores. Programmable network interface cards, for example, utilize thread-level parallelism across multiple processor cores to handle multiple events concurrently. However, the synchronization required to access a device's shared external I/O interfaces lead to scalability limitations and diminishing returns.

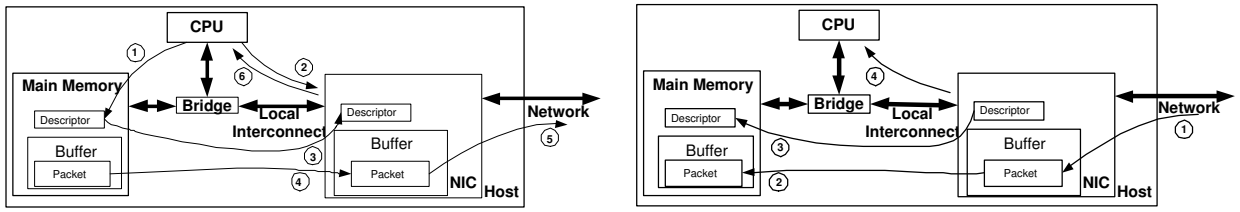
This paper compares the instruction-level parallelism versus thread-level parallelism in control-dominated network interface firmware and finds that though thread-level parallelism scales to higher levels of performance, there exists a significant amount of instruction-level parallelism that can be exploited by traditional wide-issue VLIW processors. For example, a seven-wide VLIW-based network interface architecture achieves approximately the same frame throughput as a two-way single-issue multiprocessor implementation. This motivates the use of wide-issue VLIW architectures typically used for media and signal processing workloads to extract parallelism left previously unutilized by existing thread-level approaches. This paper advocates that a combination of these two approaches should lead to even higher levels of performance than today's control-oriented embedded systems architectures.

1 Introduction

Embedded systems firmware is often structured using the event model, especially in systems whose processing is initiated primarily by external events. Network interface cards (NICs) are one such example of these types of systems, as there is only work to be done when the system sends frames or when frames arrive over the network. However, the tasks of sending and receiving frames are split up into a sequence of event handlers separated by extremely high latency operations, such as direct memory access (DMA) transfers between the NIC and the host's main memory. By partitioning frame processing into a sequence of event handlers in this fashion, tasks can be interleaved and prioritized based upon the type of event.

The event handlers in a modern network interface are typically quite short (less than 220 dynamic instructions on average) and are often control-intensive (requiring different control sequences based on what triggered the event). These event handlers form a natural unit of concurrency. Each handler performs an independent stage of frame processing that can operate in parallel with other handlers that are processing different frames. Furthermore, with the appropriate mechanisms, the same handler can operate in parallel with itself on different frames. Therefore, frame-level parallelism can easily be exploited by systems that employ the event model. However, instruction-level parallelism within the event handlers is assumed to be low because of their small size and control-intensive nature.

This paper explores the trade-offs between exploiting parallelism within an event-driven network interface using multiple processing cores versus using wide-issue very long instruction word (VLIW) processing. While parallel processing yields close to linear speed-ups for small numbers of cores, there is also a significant amount of parallelism within the event handlers. In fact, a VLIW core with 3 integer units and 3 load/store units achieves 39% higher frame throughput than a VLIW core with a single integer unit and a single load/store unit. While multiple cores provide unmatched benefits, this work motivates that these parallel cores should be more sophisticated than the single-issue ARM or MIPS cores commonly used for such event-driven embedded systems; this paper shows that a more sophisticated core can extract intra-task parallelism left previously untapped by the single-issue processor within multiple core designs.



(a) Sending a Frame

(b) Receiving a Frame

Figure 1: Steps Involved in Sending and Receiving a Frame

The rest of this paper proceeds as follows. The next section describes the operation of network interfaces. Then, Section 3 describes the hardware and software issues in parallelizing network interface firmware. Section 4 describes the experimental methodology and Section 5 presents the results. Finally, Section 6 describes related work and Section 7 concludes the paper.

2 Network Interfaces

A network server's host operating system (OS) uses its network interface card (NIC) to send and receive packets. On a programmable NIC, firmware running on one or more programmable processors must manage all interactions with the OS. These interactions include transmitting frames that the OS has queued for transmission, copying received frames into the operating system's memory buffers, and notifying the OS when such interactions have been completed. The NIC firmware is responsible for directing I/O operations by managing dedicated onboard I/O hardware that efficiently transfers data between the network and NIC memory, and between NIC memory and host memory. Because NIC I/O operations can experience latencies that last thousands of cycles, NIC firmware breaks up requests to transmit and receive frames into several processing steps. Each processing step is triggered by an event that indicates a high latency operation initiated by the previous processing step has completed. This allows the firmware to start long-latency operations, continue processing other steps, and then pick up execution again later after long-latency I/O requests have completed.

2.1 Network Interface Processing Steps

Figures 1(a) and(b) depict the host and firmware interactions necessary for sending and receiving frames, respectively. When sending frames, the host application and OS first prepare frames in host memory prior to handing them to the driver for transmission. In step 1 of Figure 1(a), the device driver creates one or more *buffer descriptors* for each pending frame. A buffer descriptor specifies the host memory address, length, and flags for a portion of a frame. If a packet is comprised of multiple non-contiguous memory regions, the device driver creates multiple buffer descriptors. After preparing buffer descriptors for a pending frame, the device driver then writes to a memory-mapped location on the NIC that indicates new buffer descriptors are available, in step 2. The firmware notices this update and then initiates one or more direct memory access (DMA) transfers to retrieve the descriptors in step 3. In step 4, the firmware inspects the buffer descriptors and initiates one or more DMA transfers to move the frame data from host memory into the NIC's transmit buffer. When the data transfer completes, the firmware directs the NIC's medium access control (MAC) unit to transfer the frame in step 5. Finally, in step 6, the firmware notifies the host OS that the frame has been sent, possibly by interrupting the host CPU.

Receiving frames is analogous to sending them, but the device driver must also pre-allocate a pool of main-memory buffers for arriving packets. The notification and buffer-descriptor retrieval process for preallocated receive buffers happens just as in the send case, following steps 1 through 3 of Figure 1(a). Figure 1(b) depicts the process of receiving a frame after preallocted buffers have arrived at the NIC. In step 1, a frame arrives via the MAC unit into local NIC memory, and the MAC unit notifies the firmware. The firmware then initiates a DMA transfer of the packet into a preallocated host memory buffer in step 2. In step 3, the firmware produces a buffer descriptor with the resulting host address and length of the received frame and then initiates a DMA transfer of that buffer descriptor to host memory. Finally, in step 4, the

firmware notifies the host's device driver that a new frame has arrived, typically via interrupt. The device driver may then check the number of unused receive buffers, allocate more, and notify the NIC.

2.2 The Event Model

Network interface firmware carries out the steps in Figure 1 using a series of event handlers. As in other event-driven software, the NIC firmware associates an event handler function with each type of event that may need to be processed; when an event arrives, the firmware's central processing loop inspects the event type and then dispatches the associated event handler to process that type of event. The event types in NIC firmware correspond to the completion of low-level hardware I/O events and of higher-level software events. The steps in Figure 1 are a combination of low-level and high-level events.

For example, when any memory-mapped NIC location is written by the host, the `mailbox` handler is invoked. This corresponds to step 2 of Figure 1(a). However, depending on the type of memory-mapped location written (that is, either transmit or receive), the `mailbox` handler will mark a different type of software event as pending. For instance, if the host has indicated that more frames are ready to be sent (as in step 2 of Figure 1(a)), then the `mailbox` handler will mark the `fetch_send_bds` event as pending. Upon return to the central event-processing loop, the `fetch_send_bds` handler will be invoked, which will enqueue a DMA read request for the corresponding send buffer descriptors to the NIC's DMA hardware; this enqueue operation implements step 3 of Figure 1(a).

Just as memory-mapped I/O writes trigger the generic `mailbox` event whose handler must decide what further processing is necessary according to the type of location being written, the completion of DMA read operations trigger the generic `dma_read_complete` event. The `dma_read_complete` handler then inspects the type of DMA that has completed. If the completed DMA is a send buffer descriptor, then the `fetch_frame_data` event will be marked as pending by the `dma_read_complete` handler so that the data corresponding to the buffer descriptors will, in turn, be fetched. This occurrence would correspond to step 4 of Figure 1(a). If, on the other hand, the DMA that has completed is of frame data, the `dma_read_complete` handler immediately directs the NIC's medium access interface to transmit the data, which corresponds to step 5 of Figure 1(a). As with the steps in Figure 1(a), the steps in Figure 1(b) also decompose into a series of events and event handlers.

The decomposition of the steps in Figure 1 into events and corresponding event handlers facilitates efficient firmware execution and avoids excessive stalling while high-latency I/O operations are pending. For example, after enqueuing buffer descriptors to be fetched via the NIC's DMA interface in step 3 of Figure 1(a), the `fetch_send_bds` handler returns to the central event-processing loop, where any other pending event can then be processed, such as an update to a memory-mapped NIC location (via the `mailbox` handler). NIC firmware prioritizes events according to type, ensuring that high-priority events will have precedence over low-priority ones. This prioritization is important to avoid exhausting finite NIC resources (such as the NIC's receive buffer) when several types of events are pending.

3 Parallelization Schemes

The goal of any software parallelization scheme is to improve application performance by executing separate parts of the application concurrently. Instruction-parallel schemes leave the control flow of the original application intact, seeking out independent instructions along the control path that can be executed concurrently. Rather than parallelizing about individual instructions, thread-parallel schemes treat threads as the fundamental unit of concurrency; consequently, applications must be carefully organized into separate threads to achieve concurrency in a thread-parallel system.

Thread-parallel schemes rely on multiprocessor or multithreaded hardware to improve performance, whereas instruction-parallel schemes require a processor that is capable of executing more than one independent instruction at a time. Multiprocessor systems tend to have increased performance at the price of increased memory system and communication complexity. Similarly, superscalar instruction parallel processors increase performance at the price of increased instruction control logic over comparable VLIW architectures. VLIW instruction-parallel architectures eliminate some of the control logic by relying on an intelligent compiler to find independent operations and group them together into a single instruction word, or bundle.

The network interface firmware used in this study is the type of workload typically parallelized across multiple processors at the frame or task level, as will be described in Section 3.1. Event handlers that dominate program runtime, as described in Section 2, tend to have rather small static and dynamic instruction counts. Handlers typically average about 95 static instructions and 217 dynamically executed instructions, many of which are broken

up over multiple short basic blocks in the code schedule. VLIW architectures have typically been avoided in this domain because of these short handlers with numerous basic blocks. This section examines both thread-level and instruction-level approaches to exploiting parallelism in NIC firmware.

3.1 Thread-Level Parallelism

As described in Section 2.2, NIC firmware events may be triggered either by an I/O occurrence, such as completion of a DMA read, or by another software event handler as a mechanism for implementing software interrupts. There are two thread-parallel approaches to parallelizing event handler execution. Task-level parallelism treats the event handlers themselves as the fundamental unit of concurrency. However, frame-level parallelism treats per-frame operations as independent, regardless of the event handler that must be executed to perform that operation.

Task-parallel firmware allows handlers of different types to execute concurrently; each thread can execute an unprocessed event as it happens. Static handler assignment avoids synchronization among threads. In such an organization, each handler type is statically assigned to a specific thread running on a specific processor. Static assignment combined with a jump-table organization for the event handlers leads to a highly efficient firmware organization. In previous work, Kim *et al.* examined and profiled task-parallel NIC firmware and found that just a few handlers dominate execution time, which would lead to load imbalances and scalability limitations with more than three threads [5]. Furthermore, the results from that study indicate that dynamic assignment would not alleviate the load imbalance and scalability problem. Profiling information shows that just a few software handlers dominate the progam execution time. Because task-parallel firmware only allows one instance of a specific handler type to execute at a time, additional handler threads would remain idle while the threads executing dominant handlers would be a bottleneck.

Frame-parallel firmware eliminates this scalability limitation by building upon the observation that frame processing for separate frames should be allowed to proceed independently and concurrently. Thus, a frame-parallel organization aims to enable higher levels of concurrency by facilitating parallel execution of multiple instances of the same handler type. Such an organization requires a more sophisticated event delivery mechanism than task-parallel firmware. In task-parallel firmware, it is sufficient to indicate only that a particular event is pending. In frame-parallel firmware, each event must also be associated with a set of frames which need processing. So, the same event may be pending multiple times for different sets of frames. Willmann *et al.* introduced a frame-parallel firmware organization and corresponding architecture that uses a software-managed distributed task queue through which events are associated with specific frames [14].

3.2 Instruction-Level Parallelization

VLIW style architectures have traditionally been used in embedded systems for media processing and numerically intensive computing workloads, such as the Texas Instruments C6x series digital signal processors, and Philips Trimedia media processors [8, 12]. In order to evaluate the effectiveness of VLIW style architectures on task-based embedded-systems workloads, this study examines the amount of instruction-level parallelism in the event-based, frame-parallel firmware described in the previous section. This evaluation was performed via an offline assembly–to–assembly instruction scheduler that takes as input an assembly listing of the firmware as compiled for the single issue MIPS R4000-based uniprocessor programmable NIC, and performs a VLIW instruction schedule for a user specified target VLIW architecture that supports the MIPS R4000 instruction set. In this study, the number of integer ALUs (arithmetic logic units), load/store and branch units in the target VLIW processor's execute stage were varied and the quality of the instruction schedule was observed. More specifically, for various VLIW instruction bundle sizes and configurations, VLIW instruction bundle utilization verses NOP (no operation machine instruction) slots is observed. Note that no floating point units are included in this study as the firmware analyzed does not include any floating point operations.

For each configuration, the instruction scheduler attempts to pack the instructions within a given basic block into VLIW bundles of a given size. VLIW bundles can contain multiple instructions that execute in parallel within a given clock cycle, or in the instance of no available instructions to schedule, NOP instructions are inserted into a given bundle which do not perform any useful computation. No intra–block instruction hoisting is performed, and the original one clock cycle branch delay slot of the MIPS R4000 processor is preserved. It should also be noted that since is this is a binary–to–binary optimization, all original register to register dependences within the original input application are preserved. That is to say, false dependences between source and destination operands of instructions in the input binary create a pes-

simistic estimate on the amount of ILP (instruction level parallelism) that can be found for the VLIW instruction schedule. Typically this is something that would be solved by building SSA (static single assignment) form using the input binary. Or, in the case of intra–block instruction scheduling using def–use chains for dependence analysis in the case that multiple live ranges occupy the same register within a basic block. Due to the non–numerically intensive nature of the input code, however, rarely is it the case that the compiler schedules multiple disjoint live ranges into the same register.

4 Methodology

The thread-level and instruction-level parallelization approaches described in Section 3 are evaluated using a combination of simulation and offline analysis of execution behavior. Architectures featuring parallel MIPS R4000 processor cores are evaluated using the Spinach simulator framework. Spinach is a library for the Liberty Simulation Environment that models the various components common to programmable embedded systems and network interfaces; a Spinach model of the Tigon-II programmable NIC has been validated against its hardware counterpart [11, 13]. Spinach simulator modules include cycle-accurate memory controllers, cache controllers, bus arbiters, DMA and Ethernet interfaces, a 5-stage MIPS R4000 processor core, and a network interface test harness. Spinach network interface simulators are expressed as a hierarchical configuration of these modules. Thread-level parallel organizations are evaluated on a Spinach model of the scalable 10 Gigabit Ethernet architecture introduced in [14]. This architecture features parallel MIPS R4000 processor cores and executes the frame-level thread-parallel firmware described in Section 3.1 to exploit maximum thread scalability.

VLIW configurations are evaluated by first re-scheduling the assembly code for the same frame-level thread-parallel firmware. The MIPS R4000 assembly code is re-scheduled at the basic-block level according to each pipeline configuration examined. For this evaluation, a perfect memory system is assumed. Each basic block is then weighted according to how frequently it executes in a uniprocessor configuration of the simulator described above. The resultant execution profile yields a speedup of the overall firmware executable based on how well the VLIW schedule compresses the source MIPS R4000 instructions into VLIW bundles. This speedup is used to calculate how much faster a given VLIW configuration can perform NIC processing and how many cycles are necessary to process one packet. It is assumed that the VLIW schedule can retire one VLIW bundle per cycle; the scheduler explicitly inserts no-ops into bundles (which are not counted as instructions toward instructions-per-cycle calculations) when no independent source instructions are available. VLIW architectures typically require more NOP instructions than a conventional single-issue architecture. However, the impact of this increase in code size can easily be minimized with well-known instruction packing techniques [8].

5 Results

Tables 1 and 2 show the performance, in terms of frames processed per thousand cycles and total IPC (instructions per clock cycle), for various VLIW and multiprocessor organizations. For each configuration, the number of processors, the combined issue width of all the processors, and the number of each type of unit is given. All configurations use in-order, five stage, pipelined MIPS processors with perfect memory systems.

The VLIW implementations presented in Table 1 achieve significant increases in performance over the single issue uniprocessor base case (shown in the first row of Table 2). The first row of Table 1 shows that even moderate VLIW pipeline widths exploit instruction-level parallelism by allowing load/store instructions to occur in parallel with integer computation instructions. As additional integer ALUs are added to the VLIW pipeline, additional load/store units are necessary to supply data to these integer ALUs.

The high IPC of the wider cores goes against the common wisdom that short, control-intensive event handlers do not have much instruction-level parallelism. However, in practice, an event handler in a network interface frequently must process multiple frames each time it is invoked. Therefore, the main loop of each event handler can efficiently be scheduled for a VLIW machine, exploiting significant amounts of parallelism. Smaller basic blocks outside of these loops with low levels of instruction-level parallelism, which are quite common in the firmware, do not account for the dominant portions of program runtime. Looking at the data in Table 1, there is an increase in IPC over the basic single-issue uniprocessor implementation as more integer ALUs and load-store units are added to the pipeline. A maximum IPC of 1.67 is achieved with an issue width of 9 instructions, at which point the amount of instruction-level parallelism within the basic blocks in the original assembly code is exhausted. Therefore, adding functional units or load-store bandwidth to the processor pipeline beyond the 9-wide configuration yields little performance improvements.

Number of Processors	*Total Issue Width*	*Integer Units*	*Load/Store Units*	*Branch Units*	*Frames per Thousand Cycle*	*IPC*
1	3	1	1	1	3.71	1.19
1	4	2	1	1	4.20	1.35
1	6	2	3	1	4.97	1.60
1	7	3	3	1	5.15	1.65
1	9	4	4	1	5.20	1.67
1	11	5	5	1	5.23	1.67
1	13	6	6	1	5.23	1.67

Table 1: VLIW Width vs. Kilo–Cycles per Bundle in Instruction Parallel Implementation

Number of Processors	*Total Issue Width*	*Integer Units*	*Load/Store Units*	*Branch Units*	*Frames per Thousand Cycle*	*IPC*
1	1	1	1	1	2.64	0.69
2	2	2	2	2	5.59	1.34
4	4	4	4	4	10.37	2.62
6	6	6	6	6	11.61	3.87

Table 2: Number of Processors vs Kilo-Cycles per Packet in Thread-Parallel Implementation

Table 2 shows that there is also abundant thread-level parallelism in network interface firmware. As the number of cores is increased, there is almost a linear increase in IPC, which leads to dramatic increases in frames processed per thousand cycles. However, there are diminishing returns with more than four processors. At that point, the performance levels out because of overheads related to providing in-order frame delivery. The firmware must keep track of the original order of each frame and deliver it to the network or host in that order. Furthermore, this leads to event dispatch and synchronization overhead, whereby multiple processors are contending for shared locks in the system and spending less time doing useful computation. As more processors are used and more frame parallelism is exploited, synchronization overheads among the threads increase as contention for access to shared global resources (such as the DMA and MAC interfaces) increase. The result is more threads and processors executing in parallel, but each thread performing less and less useful work due to time spent contending for shared resources.

When comparing the results of the two organizations, there are several interesting things to note. First, the base VLIW organization in Table 1 outperforms the uniprocessor organization in Table 2. This apparent discrepancy is due to the fact that the base VLIW organization is actually a three-wide VLIW architecture (one integer unit, one load/store unit, one branch unit) as opposed to a single-issue uniprocessor configuration thereby permitting a higher maximum IPC limit due to more available functional units within the system. When using a three-integer-unit, three-load/store-unit uniprocessor VLIW configuration with an IPC of 1.65, the VLIW architecture is able to perform comparably to a two-way thread-parallel implementation by exploiting instruction-level parallelism previously thought to be unavailable. That is to say the VLIW architecture is able to extract enough intra-thread instruction level parallelism to achieve comparable IPC rates to the multiprocessor implementation, which is exploiting inter-thread instruction level parallelism only. Finally, the most aggressive VLIW schedule in Table 1 achieves greater IPC than the two way multiprocessor thread-parallel implementation. This is primarily due to the fact that there is zero synchronization for a single VLIW core-based implementation thus the ALUs within the VLIW processor do not waste idle time polling synchronization registers like the cores in the multiprocessor implementation. However, the data throughput rates for the two-way multiprocessor thread-parallel implementation are slightly higher than that of the most aggressive VLIW schedule. This is due to the VLIW based implementation only extracting parallelism within a given basic block in a given thread. While the IPC is higher for this VLIW configuration, only one thread executes at a time. Data transfers in the system are performed not via CPU programmed I/O operations, but rather via external DMA engines that are controlled by the thread handlers. Since thread handlers control the DMA engine behavior, the lack of multiple threads executing concurrently limits the amount of data throughput the system can achieve despite an increase in overall IPC. The fact that DMA based data transfers decouple CPU performance

from total data throughput is why the thread parallel implementation, despite having a lower total IPC than the most aggressive VLIW implementation, still achieves a higher data throughput rate.

This is because the thread-parallel organization can exploit higher levels of data throughput parallelism when accessing the onboard I/O hardware due to multiple threads and handlers executing concurrently. While the most aggressive VLIW implementation achieves a slightly higher IPC, only intra-thread parallelism is exploited, and thus parallelism between threads in the system can not be extracted.

6 Related Work

Network processors (NPs) make significant use of parallel processing to accomplish network routing tasks. The two dominant parallelization approaches utilized by NPs are pipelined and threaded execution [2, 3]. However, such concurrency models are poorly suited to NIC processing, because NICs experience much larger latencies than those experienced during NP execution. These large host-to-NIC latencies lead to several hundred frames being in flight at any given time [14]. A purely threaded model would have to support hundreds of concurrent contexts, while a pipelined model would experience extreme load imbalances among stages. For example, Mackenzie *et al.* found such mismatches when using the Intel IXP network processor for network interface processing [6]. Their system used an Intel IXP 1200 operating at 232 MHz and used about 40% of its processing power to achieve 400 Mb/s of peak throughput, suggesting a peak throughput of 1 Gb/s. Though the IXP 1200 features six multithreaded cores and a StrongARM processor, the much older Tigon-II programmable NIC achieves Gigabit speeds with only two simple 88 MHz cores. This suggests that network processor-based solutions are not as efficient at handling a network interface workload as a system designed explicitly as a network interface controller. Furthermore, the multi-gigahertz cores found in router line cards dissipate far more power than is allowed or practical for a PCI card in a network server.

Previous work by the authors has explored alternate parallelization strategies for multi-core network interfaces [5, 14]. In [5], a static task partitioning was used to exploit the two processing cores of the Tigon-II network interface architecture; the performance results presented in that work show that parallelized Ethernet firmware can improve the throughput of standard IP-based protocols and applications to levels comparable to modern ASIC-based network interfaces. However, such a static partitioning is not scalable beyond approximately three processors due to load balancing constraints. In [14], frame processing was dynamically partitioned to overcome this limitation. By enabling the same task to execute in parallel with itself to process different sets of frames, such an organization can efficiently achieve 10 Gb/s line rates. In both of these studies, however, only single-issue cores were considered, thereby ignoring any potential opportunities to exploit instruction-level parallelism for improved performance.

Shivam, *et al.* have also implemented a parallelized version of their own Ethernet Message Passing (EMP) protocol on the Tigon-II [10]. EMP is a specialized message passing protocol for clusters, designed to provide a low latency and high bandwidth message passing system that is based on user-level access to Gigabit Ethernet [9]. The parallelization scheme for EMP statically partitions send and receive processing, somewhat similarly to the partitioning scheme described in [5]. However, they empirically pick a partition of major functional components among a few different partitions based on their experimental results, rather than deriving a partition that balances load and minimizes synchronization using profiles as well as careful analysis of hardware and data sharing.

There have also been several studies that show networking performance can be increased by parallelizing network protocols on general-purpose multiprocessor operating systems [1, 4, 7]. Such parallelization schemes exploit concurrency at various levels, such as across packets, protocol layers, and connections. Parallel network protocol processing deals with the layers above the network interface, so such schemes are orthogonal to the network interface parallelization strategies discussed in this paper.

7 Conclusions

The event-driven firmware of modern network interfaces exhibits multiple levels of parallelism. Despite the fact that event handlers are short and control-intensive, there is still a significant amount of instruction-level parallelism available. In fact, a 7-wide VLIW processor with three integer units and three load/store units achieves an IPC of 1.65. This results in a 39% increase in frame throughput over a base 3-wide VLIW organization, to 5.15 frames per thousand cycles. This is comparable to a two processor thread-parallel organization, which achieves 5.59 frames per thousand cycles. However, multiprocessor organizations scale to much greater levels of parallelism by allowing event handlers to process frames concurrently. For example, a six processor thread-parallel organization can process 11.61 frames per thousand cycles. One of

the major advantages of the thread-parallel organization is it's ability to exploit higher levels of memory parallelism by utilizing onboard I/O hardware in parallel. Whereas the thread-parallel organization enables separate threads to access the MAC and DMA hardware simultaneously via a memory-mapped interface, the VLIW organization is constrained to exploiting instruction level parallelism within a single basic block in a single thread.

While the instruction- and thread-parallel NIC firmware parallelization schemes have been addressed independently in this work, these two approaches are not mutually exclusive. Both approaches have points of diminishing returns. Where additional multiprocessing might be infeasible or ineffective, NIC designers should consider VLIW architectures for additional performance. Additionally, VLIW-style architectures cannot attain performance gains on par with thread-parallel implementations by exploiting instruction-level parallelism alone, but can achieve intra thread performance improvements that traditional in-order processors cannot due to their single issue pipeline structure. While single issue multicore systems provide unmatched benefits, this work suggests multicore VLIW systems whereby intra thread instruction-level parallelism is exploited at the VLIW level, while task and frame level parallelism is exploited at the thread and processor core level.

Acknowledgment

This work was supported in part by Texas Instruments, Inc., Advanced Micro Devices, Inc., and NSF under grants EIA-0224458, EIA-0321266 and CCR-0209174.

References

[1] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proceedings of the ACM SIGCOMM '93 Conference*, pages 74–83. ACM Press, 1993.

[2] L. Gwennap. Cisco Rolls its Own NPU. *Microprocessor Report*, Nov. 2000.

[3] T. R. Halfhill. Sitera Samples its First NPU. *Microprocessor Report*, May 2000.

[4] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[5] H. Kim, V. S. Pai, and S. Rixner. Exploiting Task-Level Concurrency in a Programmable Network Interface. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[6] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An Intel IXP1200-based Network Interface. In *Proceedings of the 2003 Annual Workshop on Novel Uses of Systems Area Networks (SAN-2)*, Feb. 2003.

[7] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. F. Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the Operating Systems Design and Implementation*, pages 125–137, 1994.

[8] N. Seshan. High VelociTI Processing Texas Instruments VLIW DSP Architecture. *Signal Processing Magazine*, 15(2):86–101, Mar. 1998.

[9] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001)*, Nov. 2001.

[10] P. Shivam, P. Wyckoff, and D. Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 64–69, Apr. 2002.

[11] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, November 2002.

[12] J. van Eijndhoven, F. Sijstermans, K. Vissers, E. Pol, M.I.A.Tromp, P. Struik, R. Bloks, P. van der Wolf, A. Pimentel, and H. Vranken. Trimedia CPU64 Architecture. In *International Conference on Computer Design*, pages 586–592, Oct. 1999.

[13] P. Willmann, M. Brogioli, and V. 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*, pages 20–29. ACM Press, July 2004.

[14] P. Willmann, H. Kim, V. S. Pai, and S. Rixner. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *Proceedings of the Interna-*

tional Symposium on High-Performance Computer Architecture, 2005.