ANALYSIS OF THE EFFECTIVENESS OF MULTITHREADING FOR INTERRUPTS ON COMMUNICATION PROCESSORS

Vinu Pattery, Ben Lee, Chulho Won
ECE Department
Oregon State University
{pattery, benl, chulho}@ece.orst.edu

Heui Lee
Advanced Digital Chips™ Inc.
Samkwang Bldg,
21-4 Kangnam-Ku
Seoul, Korea
hlee@adc.co.kr

Hyeong-Cheol Oh
School of Engineering
Korea University
Chochiwon, Chung-Nam, Korea
hyeong@tiger.korea.ac.kr

Abstract

High bandwidth of networks demands high performance communication processors that integrate application processing, network processing, and system support functions into a single, low cost System-On-Chip solution. However, conventional processors, when used in network related applications, are beset by the overhead of save/restore of register context, cache misses due to fetching interrupt handler from memory, and the possibility of NIC buffer overflow. Therefore, this paper analyzes the effectiveness of multithreading to service interrupts on an embedded processor simulator enhanced with a multithreaded hardware execution model. Our simulation results reveal that multithreading for interrupts from a single NIC brings only a modest improvement to processing performance. However, our analysis also shows that multithreading for interrupts has a lot of potential when applied to communication processors with multiple interrupt sources, such as Ethernet, ATM, USB, and HDLC.

Keywords: *Multithreading, UDP/IP, device driver, interrupt processing, network interface, communication processor.*

1. Introduction

The growth of the Internet is creating an increasing demand for efficient, cost effective, and low power solutions to network-enabled consumer appliances. Network appliances are highly application specific and are catered to service few tasks and thus allowing microprocessor speeds to stay low. Such appliances are built with application specific, embedded processors. However, with bandwidth of networks increasing steadily, the high speed handling of data flowing through the network is also becoming increasingly crucial. Although the network infrastructure is capable of delivering high bandwidth data, the points on the network where data is either forwarded or intercepted often pose as bottlenecks in network communication [6]. Therefore, communication processors that integrate application processing, network processing, and system support functions into a single, low cost System-On-Chip (SOC) will become crucial for efficiently handling the available data bandwidth on current and future networks.

When a communication processor receives an interrupt from the network interface card (NIC) indicating a packet arrival or completion of a packet send, it stops execution of the current application program. The state of registers is then saved and the interrupt handler code is allowed to execute. Once the handler completes its job, the context is restored back for the application program to resume execution. However, interrupts occur often and each context save/restore is costly. Therefore, this paper proposes multithreading (multiple hardware register contexts) to avoid the overhead of saving register context into memory. The effectiveness of multithreading to handle interrupts due to network packets, from the perspective of a communication processor, is evaluated.

In order to verify the effectiveness of the proposed idea, multithreading for interrupts was implemented on AE32000 processor [4] simulator called ESCASim. AE32000 is the newest 32-bit member of Advanced Digital Chips™ (ADC) Inc. EISC [5] processor family. Although there are plenty of 32-bit RISC architectures, most of them are designed around a 32-bit instruction word and suffer from poor code density for embedded applications. In order to address this code-size problem, ADC's 32-bit EISC processors use an approach similar to those of ARM's Thumb [8] and MIP's MIPS16 [9], i.e., facilitating a 32-bit data processing with an efficient 16-bit instruction coding. This approach provides better performance than a 16-bit processing scheme with better code density than a 32-bit instruction coding [8, 9]. The chip is in production and is currently used in applications such as Karaoke systems, game machines, video editors, and set-top boxes.

This paper is organized as follows. Related work in the field of exceptions and embedded processors is discussed in the following section. Section 3 discusses the operations of the device driver and its interaction with NIC and the upper layers of the protocol stack. Section 4 presents the proposed multithreaded execution model for AE32000 processor. In Section 5, simulation results and performance details are discussed. Finally, Section 6 presents a conclusion and future direction.

2. Related Work

Multithreading was originally proposed for tolerating long latency events in parallel processing systems, such as cache misses or local memory misses that require remote memory accesses [7]. This is done by executing a new thread of computation on a new hardware context, thereby overlapping memory latency with computation. Multithreading has also been proposed for modern superscalar processors to exploit thread-level parallelism (TLP). Multiple threads of execution are generated from a single program, either by the compiler or dynamically through speculation [13], and scheduled onto specially modified multiple-issue processors. Moreover, multiple threads of execution can be generated from different programs and simultaneously executed on a wide-issue processor. This technique, called simultaneous multithreading (SMT) [12], is being employed in Alpha EV8 [14], and Intel recently announced that the Xeon processor will support SMT.

While earlier multithreaded schemes were implemented to hide cache misses, Zilles *et al*. [2] proposed multithreading to hide latency due to hardware exceptions in superscalar processors (i.e., TLB misses). Thus, exploiting control and data independence by executing the exception handler in a separate hardware context.

Commercial vendors such as Intel, IBM, and numerous start-ups are employing multithreading techniques to exploit the parallelism in network workloads. For example, Intel IXP 1200 has six micro engines that have four threads each working in parallel to process network packets [16]. The six micro-engines are capable of moving, in parallel, six different incoming packets from an input FIFO buffer to a DRAM buffer. If a micro-engine's current context stalls on a memory operation, it automatically switches to a new context [16]. These designs are targeted as network processors that exploit the benefits of multithreading for memory stalls and do not specifically exploit multithreading for interrupts.

The work that comes closest to ours is the UNUM processor architecture [1]. It removes DMA and instead uses the processor core with special instructions as a means to pump data back and forth between memory and I/O devices. UNUM also employs multiple hardware contexts with priorities to handle interrupts. However, their assumption is based on 64-byte burst size leading to very frequent event processing. Thus, it is more appropriate for ATM processing.

3. Device Driver Operation

Before the proposed multithreading for interrupts on AE32000 is presented, this section discusses how the NIC driver interacts with NIC and the upper layers of the protocol stack (i.e., Socket, UDP, IP). This discussion will clarify the motivation for the proposed execution model in the following section.

The NIC driver consists of two routines: Driver Send and Driver ISR. The Driver Send routine is responsible for transferring packets to the NIC buffer, while the Driver ISR (interrupt service routine) services interrupts from the NIC as a result of a packet arrival or a packet send. The operation of Driver Send is shown in Figure 1. Driver Send initiates a transmission when called by the protocol layer, which in turn was invoked by the user application using a SWI (software interrupt). It checks to see if the NIC controller is ready to transmit by reading the command register. If ready, Driver Send using the NIC's remote DMA transfers data from the kernel memory to NIC's send buffer, then issues the transmit command and returns. If NIC is busy, Driver Send queues the packet in the transmit-pending queue of kernel memory and returns. After a transmission is initiated, Driver ISR services the interrupt from the NIC by executing the following steps (Figure 2):

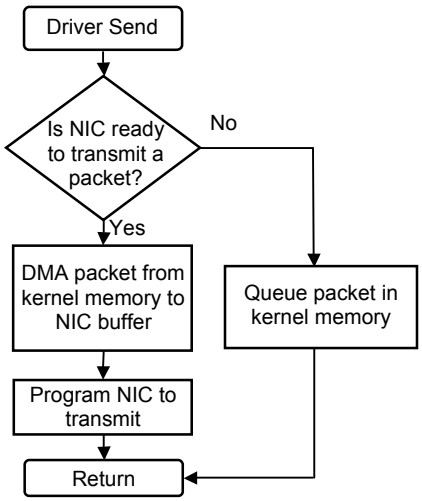


Figure 1. Driver Send Routine

(1) Reset transmit bit in the interrupt status register of NIC.
(2) Checks for successful transmission.
(3) Transmit the next packet if there are more packets in the transmit-pending queue.
(4) Otherwise, check for any pending interrupts.

In contrast to send operations, a receive operation is interrupt driven and thus resides completely within the Driver ISR routine. When a receive interrupt occurs, one or more packets may be buffered by the NIC. Packets are then DMAed from the NIC's receive buffer to the kernel memory. Packets are removed until the receive buffer is empty. The portion of Driver ISR that implements packet receive, does the following:

(1) Reset the receive bit in the interrupt status register.
(2) Remove the next packet from the receive buffer.
(3) Check to see if the receive buffer is empty.
(4) If buffer is empty, go to step (1); otherwise, read interrupt status register for any more pending interrupts.

Once Driver ISR runs to completion, SWI gets invoked and causes the processor to run in kernel mode. Since Driver ISR has completed execution, it does not have the overhead of saving register context. IP, UDP, and Socket processing occur, respectively, with each component invoking its following component by a function call. The Protocol layer performs a function return, which causes a transfer to user-level processing.

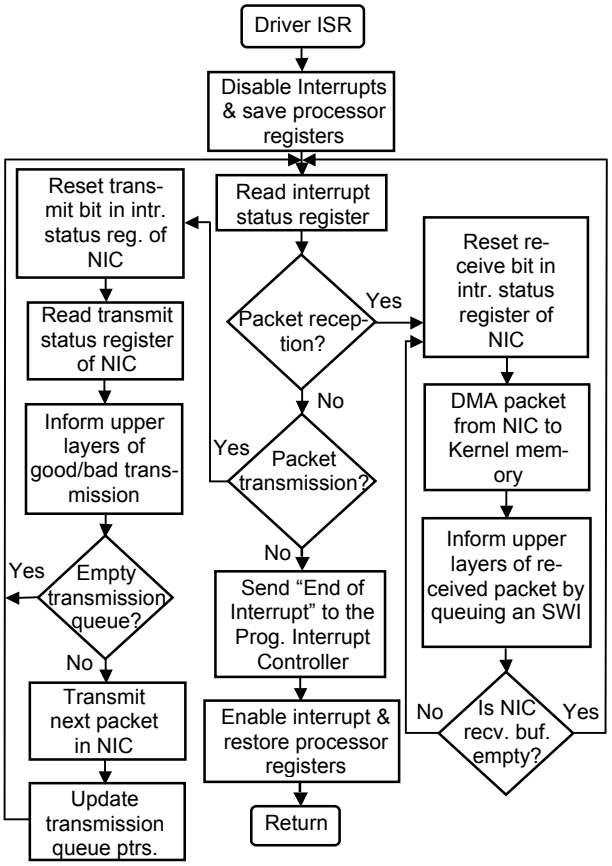


Figure 2. Driver ISR Routine

4. The Proposed Multithreaded Execution Model for AE32000 Processor

AE32000 is the newest 32-bit member of Advanced Digital Chips (ADC) Extendable Instruction Set Computer (EISC) processor family [5]. It employs a 5-stage pipeline scheme consisting of fetch (IF), decode (ID), execute (EX), memory (MEM), and write-back (WB) stages. AE32000 core is equipped with a 32-bit ALU, a 32-bit Barrel shifter, and a 32×32-bit parallel multiplier. It also has a MAC unit for DSP applications such as the DCT computation. For a detailed description of the AE32000 processor, please refer to [4].

The interrupt processing for the AE32000 processor is illustrated in Figure 3. As can be seen from the figure, when an interrupt is received, any instructions that have proceeded beyond the fetch stage are allowed to complete. Therefore, the only instruction nullified is the one in the fetch stage. Once the instruction right before the nullified instruction completes the write-back stage (and thus updates the register file), the control unit pushes the status register, resets interrupt bit in the status register, and the program counter onto the stack in the next three cycles. Then, in the following cycle, the processor's state-machine obtains the interrupt vector from the Programmable Interrupt Controller (PIC) and acknowledges its receipt by asserting interrupt acknowledge signal. Next, the interrupt vector is used to obtain the address of the interrupt handler, which is finally used to load the

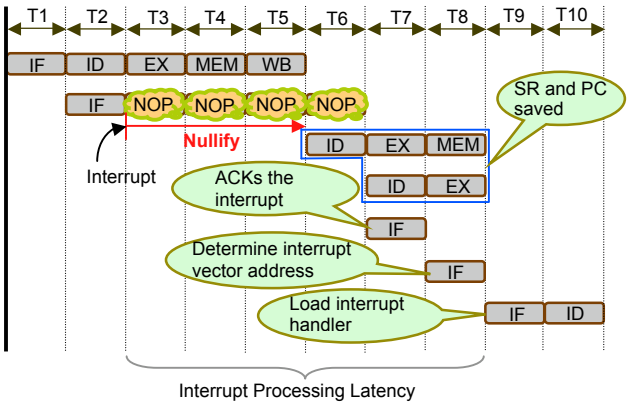


Figure 3. The AE32000 pipeline during the processing of an interrupt from NIC.

interrupt handler code in the following cycle. In all, there is a 6-cycle delay after which the interrupt service routine can be executed.

In order to implement multithreading for interrupts on AE32000, several modifications are needed. First, multiple register contexts are needed by duplicating the register file and special purpose registers, such as program counter, status register, stack pointer, and Extension Register. Second, a context switch must be performed by having a global pointer point to the new context. This can occur any time after the instruction right before the nullified instruction completes the write-back stage and before the address of the interrupt handler is latched on to the program counter. Finally, a multithreaded execution model i s needed to determine how many contexts are required and when context switching should be performed. This is heavily dependent on the behavior of the device driver and its interactions with packet send/receive operations.

Multithreaded AE32000 microarchitecture consists of three hardware register contexts C1, C2, and C3. The reasons for employing only three contexts will be explained shortly. The priority levels of operation of software code in these contexts are indicated by P1 (highest) to P3 (lowest). The rules for switching to a different hardware context are based on the interactions of the different layers of the protocol stack as discussed in Section 3. Register contexts C1, C2, and C3 are allocated to application layer, Driver ISR, and Protocol layer processing, respectively. Context C2 is assigned the highest priority P1, followed by C3 and then by C1, which has the lowest priority P3. A register context switch occurs when either a SWI or interrupt send/receive occurs. When a particular layer completes its execution, the processor execution returns to the context that initiated the context switch.

In order to illustrate the interactions among the various layers and how context switching rules are applied, Figure 4 shows the sequence of events when a receive followed by a send operation occurs during the execution of the user-level code in context C1. When a NIC (Receive) interrupt occurs, it causes the processor to switch to context C2 and run Driver ISR at priority P1. Driver ISR queues a SWI and initiates a DMA operation that moves the packet data from

the NIC buffer into the kernel memory. The SWI will not take effect immediately since the kernel is currently running at a higher priority level. When Driver ISR completes, the queued SWI causes a switch to context C3, causing the protocol layer to execute at priority P2. Then, when the NIC interrupts the processor again to signal the completion of a send initiated by an application prior to the receive interrupt, the execution of the protocol layer is stopped and the processor switches back to context C2 to run Driver ISR at priority P1. When Driver ISR completes, the pre-empted protocol layer continues execution in C3. Thus, the processor processes both send and receive packets before relinquishing the control back to the application layer. The application layer continues from where it was pre-empted in context C1.

Figure 5 shows another example where the user code running in context C1 makes a send system call. A system call in the application program code causes a SWI, which forces a switch to context C3. In context C3, the protocol layer services the system call. During this time, if the NIC has completed a transmission of packet to the network, it would interrupt the processor. The interrupt forces a switch to context C2 to run Driver ISR. After Driver ISR completes, the protocol layer resumes execution in context C3. Once the protocol layer completes, the control is then passed to the pre-empted application program in context C1.

It is important to note that Driver ISR in context C2 runs in masked mode. Therefore, Driver ISR can be run on the same hardware context since they run from start to finish without any interruption. Application and Protocol processing require two different contexts since they run on interrupt enabled mode and need to save and restore register contexts when they are interrupted. When the application layer or the Protocol processing layer receives an interrupt that requires a context switch to C2, context C2 will always be available for processing since Driver ISR always runs to completion. Similarly, context C3 can be interrupted by the NIC and cannot be interrupted by the application layer. Therefore, only three contexts are required.

5. Simulation Study

5.1 Simulation Environment

Our simulation environment consists of the AE32000 simulator, called ESCAsim, modified to support multithreading on interrupts. This was done by implementing multiple hardware contexts and an exception-handling unit. The exception-handling unit takes care of handling interrupts due to network packets based on the multithreaded execution model discussed in Section 4. The network interface is modeled by incorporating a trace execution unit into the simulator. The trace execution unit is fed with a network trace file having precise information of timing of packet arrival and departure and also time spent for processing the packet in each layers of the protocol stack.

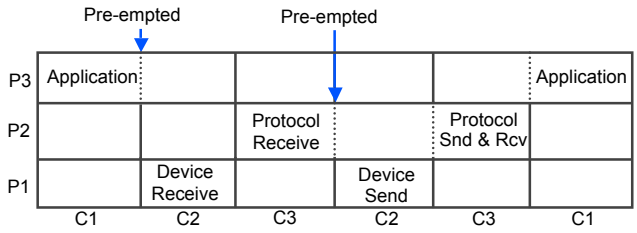


Figure 4. Processor receiving an interrupt while the application program is running.

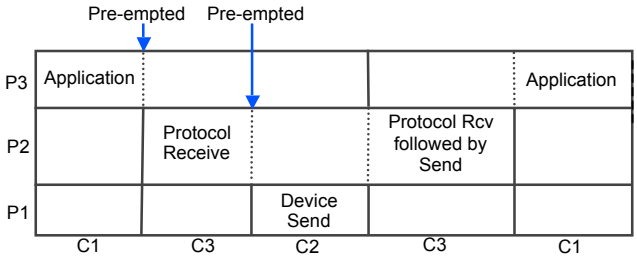
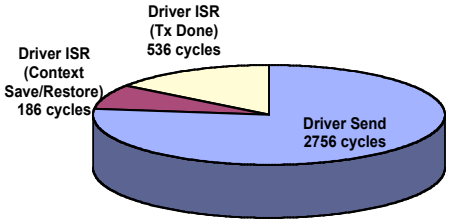


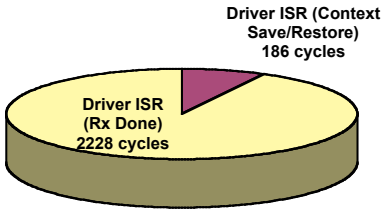
Figure 5. Application program makes a system call.

The network trace file having precise information of timing of packet arrival and departure was obtained by running *Tcpdump* network utility software on a Linux system connected to 10 Mbps Ethernet. The system call traces were obtained using the Linux tool called *Strace*. For thorough utilization of the 10 Mbps network line, a RealPlayer requesting streaming video from a RealServer was run and its traces were generated. The traces consisted of microsecond accurate information about the occurrence of system calls and packet arrival/departure. The microsecond time was then converted to relative processor clock cycles using the base processor performance as 200 MHz. The busiest one-second time interval, during which the network activity was 8.6 Mbps due to the three applications, was chosen as the required trace for the simulator. During this interval, a total of 2,732 interrupts were observed: 1,080 packet send/receive interrupts and 1,652 RealPlayer system calls (SWIs). The simulator was run for 200 million clock cycles in order to mimic a 200 MHz processor and also due to the fact that the trace file contained data for one second of operation of a 200 MHz processor.

In order to gather data related to execution time of different layers of protocol, Linux/SimOS system was employed [3]. Linux/SimOS is a Linux operating system ported to SimOS that models a MIPS R4000 processor. It is a complete machine simulator that includes all the system components, such as CPU, memory, I/O devices, etc., and models them in sufficient detail to run Linux operating system. This was done because most instruction set simulators, including the AE32000 simulator, do not model a computer system in sufficient detail to run an operating system. Without the operating system, the complete operations of the kernel cannot be simulated.



(a) Send



(b) Receive

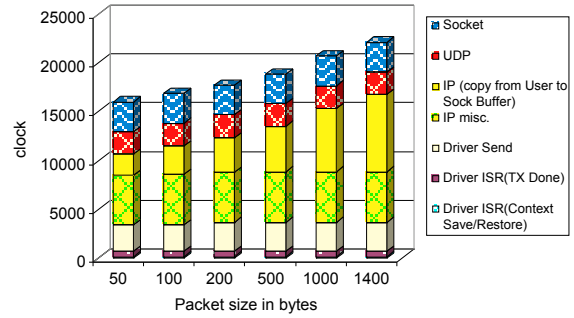
Figure 6: Device driver processing time in clock cycles for Send/Receive for packet size of 500 bytes.

5.2 Simulation Results

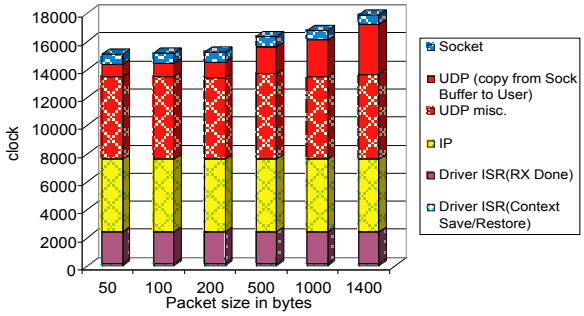
The motivation for the simulation study is to show the performance impact of multithreading for interrupts. This is done by first looking at the performance of multithreading with respect to only the driver layer. Then, the performance of multithreading for interrupts is viewed from the perspective of the entire protocol stack.

Figure 6 shows send/receive device driver processing time in clock cycles for packet size of 500 bytes derived from Linux/SimOS. The same data holds for all packet sizes ranging from 50 bytes to 1500 bytes. This is because the cost of DMAing data to/from socket buffer and NIC buffer was not included since DMA can work in parallel with the processor by cycle stealing the memory bus. It can be observed from the graphs that the device driver (Send) spends 2,756 cycles for Driver Send, this includes setting up DMA to transfer data from socket (kernel) buffer to NIC buffers and also numerous function calls to upper layers of the network. Driver ISR (Tx done) needs only 536 cycles because it is only involved in accessing NIC registers and setting the right parameters. In the device driver receive case; Driver ISR (Rx done) requires 2,228 cycles of processing time. This is because, in addition to accessing NIC registers and setting the right parameters, it sets up DMA to transfer data from NIC buffers to socket buffer and takes care of function calls to upper network layers. The processing time for save and restore of register context was measured to be 186 cycles, and remains the same for both send and receive cases.

The values from Figure 6 were then used as a parameter for multithreaded ESCAsim simulator. On a non-multithreaded ESCAsim simulator, the device driver throughput for sending a packet of size between 50 to 1500 bytes is 653 Mbps (at 200 MHz). By using multithreading, the overhead of context save and restore can be completely removed and thus achieve a data rate of 688 Mbps, which is an improvement of around 6%. In the case of packet reception, a packet size of 50 to 1500 bytes



(a) Send



(b) Receive

Figure 7. Socket to Driver layer processing time in clock cycles for Send/Receive.

results in a data rate of 994 Mbps. On the other hand, multithreading on interrupts provides a data rate of 1,078 Mbps, resulting in an improvement of almost 8%.

These results show that multithreading on interrupts results in a modest but important improvement in throughput between NIC and the driver layer. It also indicates that the improvement may be significant in the context of Network processors. By improving the data rate of the driver layer, the IP layer above it, is able to receive the data at a much faster rate. More importantly, our assumption was based on the fact that most NICs have sufficient amount of buffer space to receive an entire packet. However, low cost communication processors based on SOC environment are expected to have small on–chip buffers, which will result in increased number of interrupts and thus further improving the performance of the driver layer [1].

In contrast to driver layer processing, the performance of the entire protocol stack is relevant when communication processors are used to facilitate network-enabled consumer appliances, such as set-top boxes. Therefore, Linux/SimOS was used to obtain the relevant processing times of the various layers of the protocol stack, which was then associated with each packet based on its size. Figure 7, shows the send and receive Socket to device driver processing time in clock cycles, derived from Linux/SimOS. The processing times for the Socket layer, which mainly involves pointer and data structure manipulations, stays relatively constant through both send and receive cases. In the case of a send (Figure 7(a)), IP layer processing time is higher because it is involved in copying data from user buffer to socket buffer. The device driver processing time constitutes 22% of the overall processing

time from Socket through device driver. In the case of a receive (Figure 7(b)), UDP processing time is higher because it is involved in copying data from socket buffer to user buffer. The device driver processing time represents 13% of the overall processing time from Socket through device driver. For moving data to/from socket buffer and NIC buffer, Driver Send ISR initiates DMA operation, but no processor cycle is wasted in the actual copying of data. Again, this is because DMA operations can be overlapped with the processor executing the protocol stack. This is a reasonable assumption since our interest is in throughput performance and not the critical path performance of a single packet send or receive.

The network trace file with its associated processing time parameters was then fed to the multithreaded ESCAsim. On a non-multithreaded ESCAsim simulator, the throughput for receiving a packet of size between 50 to 1500 bytes is 8.57 Mbps; where as, multithreaded ESCAsim achieved a data rate of 8.6 Mbps between the NIC and the application layer code, which is a 0.34% improvement. Similar performance was observed for the packet-send case. Thus, it is evident that multithreading provides minimal improvement on the performance of communication processors when used in applications requiring the entire protocol stack.

Although our results were less promising when the entire protocol stack was involved, multithreading resulted in improved performance to the driver layer processing. The idea of eliminating register save/restore overhead through multithreading for interrupts has potential with regard to how communication processors will be deployed. Communication processors implemented as SOC solutions that integrate many common interface units, such as Ethernet, ATM and USB, experience higher event processing [15]. Since they have multiple network interfaces, each interface can afford to have only a very small buffer of its own. With the vast amount of network activity that can be expected in these systems, there is a need for an efficient SOC processor architecture that would maximize the performance of interrupt processing and buffer management.

6. Conclusion and Future Work

In this paper, multithreaded execution model for efficient handling of interrupts was introduced. Using our simulation model, the effectiveness of multithreading for interrupts was studied from the point of view of a communication processor. Our simulation results reveal that the concept of multithreading for interrupts brings a modest improvement to communication processor performance when limited to servicing a single source of interrupt. However, our analysis also shows that multithreading for interrupts has a lot of potential when applied to communication processors with multiple interrupt sources. Our future plan is to implement and study the performance of such a system.

7. References

[1] C. D. Cranor, R. Gopalakrishnan, and P. Z. Onufryk, Architectural Considerations for CPU and Network Interface Integration, *IEEE Micro*, *Vol. 20*, No. 1, January 2000, pp. 18-26.

[2] C. B. Zilles, J. S. Emer, and G. Sohi, The use of multithreading for exception handling, *32nd Annual International Symposium on Microarchitecture (MICRO'32)*, Haifa, Israel, November 1999, p. 219.

[3] C. Won, B. Lee *et al*., Linux/SimOS - A Simulation Environment for Evaluating High-speed Communication Systems, *Proc. of International Conference on Parallel Processing (ICPP'02)*, Vancouver, Canada, August 2002, pp. 193-202.

[4] H.-C. Oh *et al*., AE32000: An Embedded Microprocessor Core, *Proc. of the 2nd IEEE Asia Pacific Conference on ASIC (AP-ASIC'00)*, Cheju Island, Korea, August 2000, pp. 255-258.

[5] H. Lee, P. Beckett, and B. Appelbe, High-Performance Extendable Instruction Set Computing, *6th Australian Computer Systems Architecture Conference (AustCSAC'01)*, Queensland, Australia, January 2001, p.81.

[6] P. Crowley *et al*., Characterizing processor architectures for programmable network interfaces, *Proc. International Conference on Supercomputing (ICS'00)*, Santa Fe, New Mexico, 2000.

[7] R. Thekkath and S. J. Eggers, The Effectiveness of Multiple Hardware Contexts, *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, California, October 1994, pp. 328-337.

[8] *An Introduction to Thumb*, Advanced RISC Machines Ltd., March, 1995.

[9] K. D. Kissell, MIPS16: High-density MIPS for the Embedded Market, *Proceedings of Real Time Systems*, 1997.

[10] G.Y. Cho, A Study on Extendable Instruction Set Computer 32 bit Microprocessor, *J. Inst. of Electronics Engineers of Korea*, *Vol. 36-D*, No. 5, May 1999, pp. 11-20.

[11] N. Shah, *Understanding Network Processors*, version 1.0, UC Berkeley, Sept. 2001.

[12] D. Tullsen, S. J. Eggers, and H. M. Levy, Simultaneous Multithreading: Maximizing on-chip Parallelism, *22nd International Symposium on Computer Architecture (ISCA'95)*, June 1995, pp. 392-403.

[13] H. Akkary and M. A. Driscoll, A Dynamic Multithreading Processor, *31st Annual International Symposium on Microarchitecture (MICRO'31)*, Dallas, Texas, Dec. 1998, pp. 226-236.

[14] J. Emer, EV8: The Post-Ultimate Alpha, *Conference on Parallel Architectures and Compiler Technology (PACT'01)*, Barcelona, Spain, Sept. 2001.

[15] GlobalspanVirata Inc., *Helium™ 200 Communications Processor*, February 2002.

[16] T. Spalink, S. Karlin, and L. Peterson, *Evaluating Network Processors in IP Forwarding*, Princeton University Technical Report, November 2000.