Architectural Considerations for CPU and Network Interface Integration

C. D. Cranor, R. Gopalakrishnan, P. Z. Onufryk

AT&T Labs - Research
Florham Park, NJ 07932, USA

Abstract

The popularity of the Internet and the emergence of broadband access networks is fueling the development of communications processors — devices that integrate processing, networking, and system functions into a single, low-cost, system-on-a-chip. These devices form the core of cable modems, ADSL modems, set-top-boxes, smart packet phones, and a variety of Internet appliances. The conventional approach for designing communications processors is to start with a standard CPU core such as MIPS or ARM, add to this several network interfaces, and tie the whole system together with a multi-channel DMA controller. In such designs the DMA controller often becomes the most complex part of the system. The integration of processing and networking into the same device offers an opportunity to rethink the way the CPU and network interface interact. In this paper we describe UNUM, an architecture for integrating communications functionality into the CPU that not only simplifies the design of communications processors, but also improves their performance and provides them with greater flexibility.

1 Introduction

The growth of the Internet is creating a demand for broadband access equipment and network enabled consumer appliances. At the heart of these products are communications processors — devices that integrate processing, networking, and system support functions into a single, low-cost, System-On-Chip (SOC). The primary challenges in the design of these devices are minimizing cost and time-to-market, and maximizing flexibility. Die size and packaging are the major factors that determine cost. The rapid pace at which Internet applications and services are evolving increases the pressure to reduce development time, and thus designers of communications processors are continually looking for ways to speed design and verification. This rapid rate of change is also increasing the importance of flexibility. Communications processors are often adapted to applications that may not have been anticipated, or even existed, when the chip was architected.

A large body of research and experience in the design of network adaptors for workstations exists [5,6,9,14,19]. It may appear that a communications processor consists of nothing more than integrating these designs on a chip. However, this is not the case since the system requirements and constraints for workstations and communications processors are fundamentally different.

Latency in communications processors must be carefully managed to reduce on-chip buffering. Network Interface Cards (NICs) typically used in workstations plug into I/O buses that have high latencies. This forces the NIC to include large buffers and encourages large burst transfers for efficiency. For example, a 10/100 Mbps Ethernet NIC can have as much as 12KB of buffering [2]. Since communications processors often contain multiple network interfaces, placing such large buffers on-chip may not be possible and certainly does not lead to cost-effective designs.

Space requirements in workstations are not as stringent as in the SOC environment of communications processors. This allows workstation NICs to include considerable processing power. For example, “intelligent” NICs have been designed that contain on-board processors [9,10,14]. Even “dumb” workstation NICs are actually quite intelligent. For example, it is common for a NIC to contain a complex DMA controller and buffer management unit. In a communications processor this functionality is typically shared among multiple network interfaces to reduce die size. Network interfaces in these devices consist simply of a data link interface and buffers.

The integration of processing and networking in the same device offers an opportunity to rethink the way CPUs and network interfaces are designed. Most communications processor CPU cores use Instruction Set Architectures (ISAs) that were initially developed for workstation processors and optimized for SPEC like benchmark performance. Network adaptor research has been focused on reducing memory copies and host CPU processing [4,6,7,8,10,20], both of which lead to complex interface specific hardware which is not appropriate for communications processors.

In this paper we introduce UNUM, an architecture for communications processors that supports extremely fast event processing and high performance data movement. With these capabilities, functions typically performed in custom hardware can be moved to software executing on the main CPU. In the next section we review design approaches used in current communications processors. In Section 3 we describe the UNUM architecture, and in Section 4 we evaluate its performance. In Section 5 we present our conclusions.

2 Design Approaches

The design of a communications processor rarely begins from scratch. Cores are either licensed from external Intellectual Property (IP) vendors or are available from internal sources. With the emergence of on-chip bus standards (e.g., AMBA

[3], CoreConnect [18], and IP Bus [12]) and a thriving intellectual property industry, it would appear that a communications processor could be rapidly designed by licensing standard CPU and network interface cores and tying the whole system together with a multi-channel DMA controller. It has been our experience, as well as that of others, that the design and verification of this type of DMA controller is both complex and time consuming. This is especially true when features necessary for high performance such as unaligned transfers and cache-coherency are incorporated.

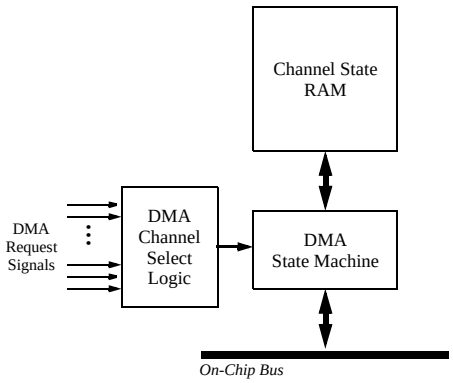

Figure 1. Multi-Channel DMA Controller

Figure 1 shows a simplified block diagram of a typical multi-channel DMA controller. Since only one DMA channel can be active on a bus at any given time, die area can be saved by designing a single DMA state machine that is shared by all DMA channels. Since there is considerable state information associated with each DMA channel (e.g., source address, destination address, byte count, and descriptor pointer) this state is commonly stored in a RAM rather than individual registers to reduce chip area. When a DMA channel becomes active, its state is transferred from RAM to the DMA state machine. After the operation completes the updated channel state is written back to RAM. Arbitration logic determines which DMA channel is serviced next.

Unlike other parts of a communications processor for which standard cores are readily available, the DMA controller must often be designed from scratch. This is because the DMA controller is highly system dependent and its design is influenced by system performance requirements, on-chip bus architecture, memory controller design, as well as the type and number of supported network interfaces.

Diversity in network interface requirements has the greatest impact on DMA design. In addition to transferring data, high performance, descriptor based, DMA controllers also transfer control and status information between DMA descriptors in memory and a network interface. This allows the DMA controller to execute sequences of transfers autonomously. The format and content of these descriptors typically needs to be modified for each type of network interface. Also the basic function of a DMA channel itself may need to be modified. For example, the destination address for a received ATM cell is not the address of a buffer pointer in a descriptor as it would be for an Ethernet frame. Instead, it is the address of a reassembly buffer which is dependent on the virtual circuit specified by the VPI/VCI fields in the cell's header. We call this form of channel customization *interface specific processing* since it requires functionality beyond simple data movement. Other examples of interface specific processing are multiplexing/demultiplexing of data based on time-slot for a TDM bus, and searching through multiple DMA descriptors in search of an "optimal" sized buffer for storing a received Ethernet frame.

Despite the complexity of designing a multi-channel DMA controller, a number of communications processors have used this approach. The AMD Am186CC [1] has a 12 channel DMA controller. Four of the channels are simple general purpose DMA channels while the remaining eight channels support descriptor based operation and are specialized for use with either HDLC or USB. The NetSilicon Net+ARM processor [15] contains a 10 channel DMA controller that supports descriptor based operation. All of the DMA channels are identical except for channel one which is customized to hunt through four pools of DMA descriptors in search of an "optimal" sized buffer for received Ethernet frames. A unique approach was used in the Euphony processor [11,17]. Rather than customizing one of the five general purpose DMA controller channels, a separate control block processor was designed that performs interface specific processing for ATM and uses an unmodified general purpose DMA channel to transfer data.

The design of a multi-channel DMA controller with the features necessary to support multiple network interfaces can be as complex as a programmable processor. For this reason, some designers have chosen to replace multi-channel DMA controllers with a dedicated processor for data transfers and interface specific processing. This eliminates the complexity of designing the DMA controller, provides flexibility, and allows modifications and enhancements to be made in software. Motorola uses a special purpose microcoded processor in its QUICC and PowerQUICC communications processor families. The microcoded processor in the Motorola MPC860 [16] uses two physical DMA channels to provide 16 virtual DMA channels, each of which is dedicated to a network interface, and two general purpose DMA channels. In addition, the processor contains DSP capabilities which can be used for applications such as a V.34 data pump. The Virata Helium processor [13] contains two ARM processors and no hardware DMA channels. One of the ARM processors is dedicated to performing data transfers and interface specific processing for ATM, HDLC, Ethernet, and USB interfaces. Code for this processor resides in an 8 KB microcode RAM that replaces the instruction cache.

Adding a second processor for data transfers and interface specific processing eliminates the complexity of designing a DMA and provides flexibility, but it also introduces the soft-

ware complexity and partitioning issues associated with developing code for multiple processors. This is especially true if the architecture of the processor that handles communications tasks differs from that of the main CPU. Since the functions of a processor must be replicated in this approach (e.g., two BIUs, two ALUs, etc.) it may increase die size. This approach also leads to a static partitioning of functions onto processors. Idle cycles on one processor cannot be used to enhance performance of tasks running on the other.

We note that applications with low data rates that can tolerate high latencies without requiring large on-chip buffers do not require a DMA controller or a dedicated processor. Instead, operations may be performed by an interrupt handler running on the main CPU. This approach is used in the T.sqware TS702 [21] which contains an enhanced SPARC processor that performs data transfers and interface specific processing for four serial interfaces.

3 The UNUM Architecture

It is our belief that a communications processor for low-cost consumer applications should contain a single processor that performs all data movement, interface specific processing, and application processing. This is especially true as embedded processors reach speeds of 500 MHz and higher. The availability of processor cores capable of performing these tasks would reduce communications processor design and verification time, increase their flexibility, and simplify software development. The two key features of UNUM, an architecture for this type of processor core, are described below.

3.1 Multithreaded CPU for Event Processing

Performing data movement and interface specific processing on the same CPU as applications processing dramatically increases the number of processor events that must be serviced. The key to minimizing communications processor cost is minimizing die size, which means minimizing on-chip buffering. Small on-chip buffers impose tight constraints on acceptable event service latency and result in small burst transfers thus increasing the number of events.

To illustrate the importance of minimizing event service latency consider a “cut-through” transfer of a 1518 byte Ethernet frame from a receive FIFO to memory. Using a burst transfer size of 64 bytes results in 24 data transfer request events. To prevent overflow, the receive FIFO must be large enough to accommodate the worst case event service latency. A large event service latency not only reduces the maximum throughput, but also requires larger FIFOs to prevent overflow. This, in turn, results in higher queuing delays which further increases worst case event service latency.

Current processors service external events using either polling or interrupts. Infrequent polling results in large event service latencies, while frequent polling consumes large amounts of processing. Both of these are unacceptable. Table 1 shows interrupt performance of some popular embedded processors. In this table, response time refers to the time from event occurrence to the start of event service routine execution, while exit time is the time to exit an event service routine. Response and exit times together represent overhead. Worst case performance must be considered during system design since taking the best or average case will lead to conditions such as buffer overflow/underflow.

System Configuration	Response Time	Exit Time
Interrupt under VxWorks on 33 MHz IDT R3081	24 µs	45 µs
Interrupt under VxWorks on 100 MHz IDT R4700	2.8 µs	8.8 µs
Interrupt under VxWorks on 220 MHz StrongARM	2.1 µs	1.2 µs
200 MHz MIPS processor, 100 MHz SDRAM Minimum context saved, no RTOS overhead	0.1 µs to 0.6 µs	30 ns to 0.3 µs
200 MHz MIPS processor, 100 MHz SDRAM Full context saved, no RTOS overhead	0.3s µs to 4.7µs	0.2 µs to 3.4µs

Table 1. Typical Embedded Processor Interrupt Performance

While performance for a particular system configuration and operating system may vary, Table 1 shows that response time for an interrupt with a full context save for high performance processors is on the order of several microseconds. A major component of response and exit times is saving and restoring the state of the interrupted context. Techniques used to reduce this overhead include: coding interrupt service routines in assembly language to use a small number of registers, switching to an alternate register set for interrupt processing, saving processor registers to unused floating point registers, and providing on chip memory for saving and restoring state. Even if interrupt overhead were eliminated, the overhead of loading and updating event service routine state from memory would still remain. This is because interrupt service routines do not retain state across invocations. A data transfer event service routine must load the starting address, byte count, destination address, and possibly a descriptor pointer on entry, and update the byte count on exit.

To eliminate the latency and overhead of interrupts, UNUM employs multiple hardware contexts with priorities. In addition, by allowing the state of an event service routine to be preserved in a CPU context across invocations, the overhead of retrieving and updating event service routine state may be eliminated. A block diagram of a UNUM processor is shown in Figure 2. It consists of three major components: an event mapper, a context scheduler, and a CPU pipeline.

The function of the event mapper is to initiate event service routine execution on occurrence of an external event. Associated with each possible external event are event mapper registers that contain the context, address, and priority of the corresponding event service routine. When an event occurs, the event mapper uses this information to initiate event service routine execution by setting the program counter and priority of the corresponding hardware context to that of the event. In cases where multiple events occur simultaneously,

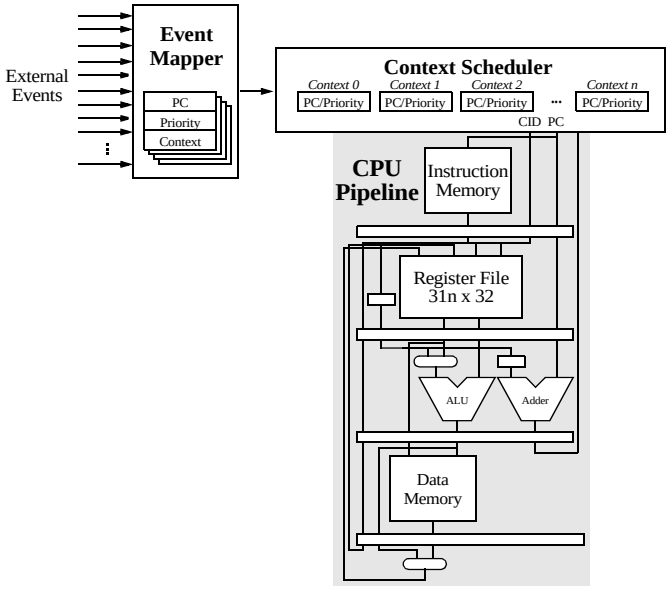

Figure 2. Example UNUM CPU

or multiple pending events map to the same hardware context, the event mapper uses the priority to determine order of invocation.

The context scheduler is responsible for issuing instructions to the CPU pipeline. Each cycle the context scheduler examines the priority of all active contexts and issues the next instruction from the context with the highest priority. In cases where multiple active contexts share the highest priority, instructions are issued from these contexts in an interleaved round robin manner.

The UNUM pipeline is a simple single issue RISC pipeline augmented to support concurrent execution of instructions from multiple contexts. The 31 x 32 register file of a typical RISC processor is expanded to a 31*n* x 32 register file, where *n* is the number of supported hardware contexts. When the context scheduler selects a context from which to issue an instruction, it presents the CPU pipeline with a Context ID (CID) and a Program Counter (PC) value. The CID together with a register number from the fetched instruction form the actual address of the register in the register file. The CID is also used by pipeline bypass and interlock logic. Thus, aside from modifying the instruction issue logic, expanding the register file, and adding a CID to bypass and interlock logic equations, UNUM employs a traditional single issue RISC pipeline.

3.2 Data Movement Instructions

General purpose processors have historically had poor data movement capabilities. Programmed I/O (PIO) generates memory-to-memory transfers which require twice the bus bandwidth of fly-by DMA operations. Some system designers have used special hardware to perform fly-by transfers as a side effect of address ranges, but this leads to complex software and does not scale well to multiple interfaces. PIO operations using non-cacheable loads and stores result in single word data transfers that achieve poor bus utilization. Using cacheable loads and stores to generate burst transfers result in data cache pollution, while using block loads and stores present in some processors increases register pressure. Unaligned PIO operations are extremely inefficient. This is especially true when transfers must be performed to an aligned fixed width memory device, such as a FIFO port. Finally, PIO operations tie up the CPU.

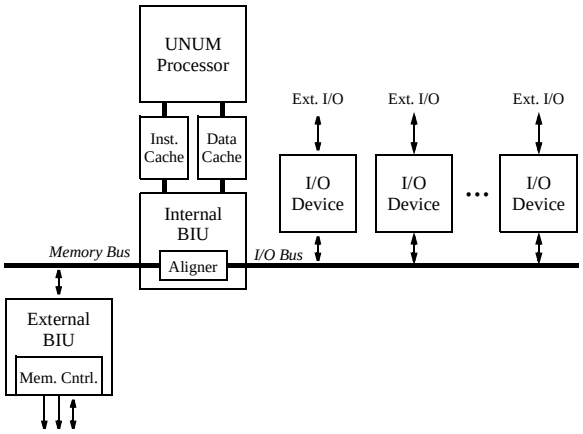

Figure 3. UNUM Based Communications Processor

Since data movement is one of the primary functions of a communications processor, data movement instructions have been incorporated in UNUM. The system architecture of a communications processor based on UNUM is shown in Figure 3. The CPU core interfaces to the rest of the system through an Internal Bus Interface Unit (IBIU). In addition to performing the operations of a traditional bus interface unit, the IBIU incorporates a data mover and aligner that segments the on-chip bus into a memory bus and an I/O bus. The CPU initiates a data movement operation by issuing an instruction to the data mover. Since data movement fully utilizes on-chip buses, an implementation may stall the CPU pipeline until the operation completes, or allow the pipeline to continue execution from on-chip caches as long as there are no misses.

Table 2 lists UNUM data movement instructions. Although each instruction performs a different operation, all instructions share the same three register format. Register *ca* contains the address of where the data transfer should begin, *bc* contains the number of bytes to transfer, and *dv* specifies the network or I/O interface. On completion of the data transfer, *ca* is updated with the address of the last byte transferred plus one. Although *bc* specifies the maximum length of the data transfer, cases exist where a transfer may be prematurely terminated by an interface (e.g., at an end-of-packet). In these cases the number of actual bytes transferred may be determined from the final value of *ca*.

UNUM data movement instructions perform fly-by transfers between memory and devices on the I/O bus. TM2D transfers data from memory to an interface while TD2M transfers data in the opposite direction. In both cases, fly-by data bypasses and does not pollute the data cache. In order to maintain

Instruction	Description
`TD2M ca, bc, dv`	**Transfer from Device to Memory**. Register *bc* contains the number of bytes to transfer from the interface specified by register *dv* to memory starting at the address contained in register *ca*. Hits in the data cache result in invalidations during the transfer. At completion of the transfer, *ca* is updated with the address of the last byte transferred plus one.
`TD2C ca, bc, dv`	**Transfer from Device to Cache**. Same as TD2M except that data is moved to the cache rather than memory. Loaded cache blocks are marked dirty. Cache replacements result in cache write-backs.
`TD2MC ca, bc, dv`	**Transfer from Device to Memory and Cache**. Same as TD2M except that non-dirty blocks in the cache are replaced with data from the device.
`TM2D ca, bc, dv`	**Transfer from Memory to Device**. Register *bc* contains the number of bytes to transfer from the starting address contained in register *ca* to the interface specified by register *dv*. Data is supplied by the data cache for dirty blocks. At completion of the transfer, *ca* is updated with the address of the last byte transferred plus one.
`TM2DD ca, bc, dv`	**Transfer from Memory to Device and Discard**. Same as TM2D except that dirty cache blocks are marked invalid and a write-back is not performed.

Table 2. UNUM Data Movement Instructions

cache consistency, dirty data is supplied by the cache during TM2D and cache invalidates are performed during TD2M. Efficient processing of network data is supported through direct network interface and data cache transfers. TD2C loads data directly into the data cache from an interface, eliminating an unnecessary transfer through memory. TM2DD allows dirty data written to a network interface to be discarded from the data cache, potentially eliminating an unnecessary future write-back.

All data flowing between the memory and the I/O buses passes through an aligner in the IBIU. For aligned transfers, the aligner simply passes data from one bus to another unmodified. For unaligned transfers, the aligner uses a holding register, shifter, and mux to align data as it flows from one bus to the other. An example of this for a 4 word unaligned transfer is shown in Figure 4.

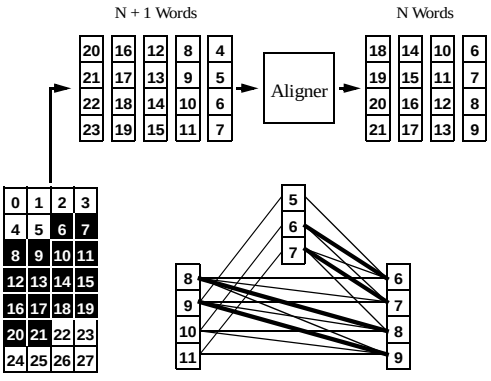

Figure 4. Example of Unaligned Fly-by Transfer

3.3 Putting it All Together

The ability to service external events with extremely low overhead together with high performance data transfer instructions allows UNUM to perform data movement and interface specific processing functions in software. This allows communications processors to be rapidly constructed by combining a UNUM processor core with network interface cores. A typical high-speed network interface in UNUM maps to two processor contexts, one for input processing and one for output processing. Threshold logic in the output FIFO of the network interface generates an event whenever there is room for an output data transfer. Similarly, threshold logic in the input FIFO generates an event whenever enough data exists for a complete data transfer or an end-of-packet is detected. The event handling routines may perform interface specific processing.

4 Simulation Results

In order to better quantify the benefits of the UNUM architecture on data movement and interface specific processing, we created a cycle-accurate simulator of a UNUM based communications processor. The CPU in the simulator was based on the MIPS32 ISA which was enhanced to support multiple hardware contexts and data movement instructions. The simulator also modeled the caches, memory system, counter/timers, a console, and an ATM interface. The modular architecture of the simulator allows us to easily plug-in other interfaces in the future.

The flexible structure of the simulator allowed us to experiment with various CPU speeds, cache configurations, and memory systems. For the measurements reported here we simulated a 200MHz UNUM processor with an 8KB 2-way set associative instruction cache, 2KB 2-way set associative data cache, and a 4 word write buffer. We configured our simulated 32-bit system bus to run at 100MHz and the memory system to consist of 100 MHz SDRAM. All of the benchmarks were written in C and compiled with an enhanced MIPS GCC 2.8.1 compiler with “-O3” optimization enabled. No hand assembly language optimizations were performed other than the ones mentioned below. In addition, we assumed that the event/interrupt handlers were locked in the instruction cache.

4.1 Data Movement

For our initial measurements we wrote a data movement micro benchmark that simulated the transfer of a 1518 byte packet from memory to a network interface using a range of burst transfer sizes and measured the resulting bandwidth. We examined two data movement mechanisms, one using UNUM data movement instructions and another using PIO. Our PIO function moved data using an optimized hand-coded assembly routine based on the BSD `bcopy()` function. We examined three CPU configurations: UNUM (hardware context switch with state preservation), fast interrupts (alternate register set with no interrupt state preservation), and normal processor interrupts. For normal interrupts we assumed an overhead of 1µs. We ran our benchmark for best and worst

case data cache scenarios for state information and with the assumption that data to be moved is not present in the data cache.

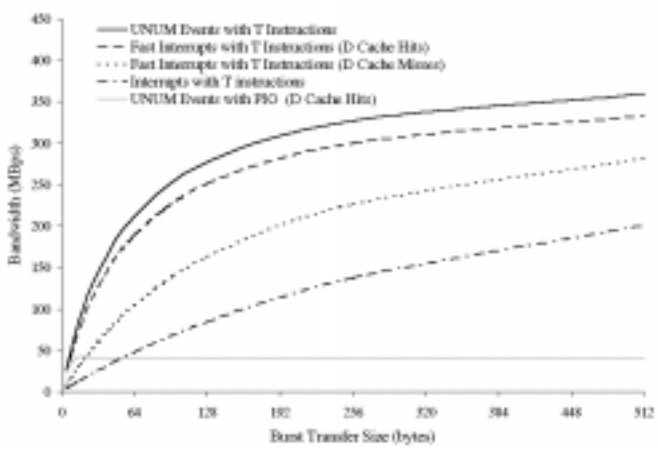

Figure 5. Data Movement Performance

The results of the data movement benchmark are shown in Figure 5. PIO-based data movement results in the worst performance. The highest achievable bandwidth using PIO was 40 MBps. This was true regardless of the type of CPU used (UNUM, fast interrupts or regular interrupts) and caching assumptions. This is because the cost of PIO dominates all other overheads.

Making use of UNUM's data movement instructions improved results significantly. For an SOC environment with small on-chip buffers we expect burst sizes in the range of 64 bytes. Using this burst size, a normal interrupt-based system achieved 47 MBps. Data cache misses had little effect since we assumed a fixed interrupt overhead of 1µs which dominiated. Replacing normal interrupts with fast interrupts improved performance to 104 MBps assuming all data cache misses, and 189 MBps assuming all data cache hits. Given the small size of data caches in SOC communications processors, we expect the actual achieved bandwidth to be closer to the lower end of this range. UNUM with data movement instructions produced the best results: 212 MBps. Since the state of the event service routine fits within a UNUM context, no state information needs to be loaded from memory. This explains why UNUM outperforms fast interrupts with data cache hits, and it also is the reason why the UNUM curve is unaffected by data cache misses.

It is also interesting to note that for very small burst sizes, UNUM events with PIO outperforms fast interrupts with data movement instructions. This is because for small bursts the overhead of loading the event service routine state exceeds that of performing memory to memory PIO transfers.

4.2 ATM Soft-SAR

Our second benchmark measures the ability of UNUM to perform complex interface specific processing. For this benchmark we selected ATM AAL5 Segmentation and Reassembly (SAR) since it represents a class of applications where the processing performed on received data is dependent on its content.

ATM AAL5 SAR transmit processing consists of segmenting Protocol Data Units (PDUs) to be sent on an ATM virtual circuit into fixed length cells and attaching a header to each cell. The PDU is padded to contain an integral number of cells, and the last cell has fields which indicate the data length, a user-to-user byte, and a CRC-32. SAR receive processing consists of reassembling received cells back into PDUs, checking the length and CRC-32 fields, and passing the payload to upper layers.

What makes SAR processing challenging is that for each received ATM cell considerable work must be performed. First, the VPI/VCI in the cell header is used to lookup the virtual circuit the cell belongs to. This lookup returns a data structure which contains a pointer to a reassembly buffer and current CRC-32 for the packet being reassembled. The payload of the cell is then appended to the reassembly buffer and the reassembly buffer pointer and CRC-32 are updated. Additional processing is required to handle boundary conditions such as end-of-frame and end of buffer. Due to the complexity of SAR processing, most systems implement this in custom hardware.

The ATM interface in the system we simulated consisted of a physical layer interface (e.g., Utopia), a transmit and receive FIFO, a CRC-32 calculator, and control and status registers. The interface generates an ATM receive event when a cell is present in the receive FIFO and an ATM transmit event when space exists for a cell in the transmit FIFO.

Code was written for performing SAR processing on the UNUM processor. The SAR software uses three hardware contexts. The first performs ATM receive event processing, the second performs ATM transmit event processing, and the third performs ATM transmit cell scheduling. Unlike some toy benchmarks, the UNUM ATM Soft-SAR represents a complete SAR including buffer management and an interface compatible with commercially available ATM signaling and ILMI software.

Our first experiment measured the maximum achievable throughput assuming an infinite line rate and FIFOs. Figure 6 shows the throughput as a function of AAL5 frame size for half duplex transmit, half duplex receive, and full duplex operation. In all three cases, the throughput increases with frame size since the per frame overhead is amortized over a larger number of cells. The highest throughput we observed was 570 Mbps which occurs for the half duplex receive case. Transmit throughput is lower because of the extra overhead associated with cell scheduling. These results show that low overhead event processing and high performance data movement instructions allow UNUM to sustain a very high throughput. As a comparison, to sustain a receive throughput of 570 Mbps a single context CPU would have to service an interrupt every 750ns.

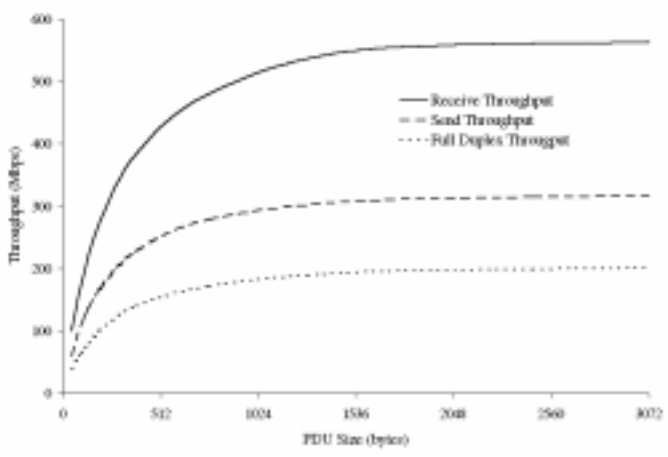

Figure 6. Maximum ATM SAR Throughput

Our second experiment measured UNUM processor utilization and FIFO size necessary to sustain a full duplex line rate of 25 Mbps. We measured a CPU utilization of 13.4% with a frame size of 1536 bytes. This means that even when transmitting and receiving at full line rate, 86% of the CPU is available for other processing. More important than throughput is worst case latency. Worst case latency determines required on-chip buffering. Using UNUM, a 25 Mbps full duplex line rate requires just a four cell transmit and receive FIFO.

5 Conclusions

In this paper we introduced UNUM, an architecture for communications processors that addresses the challenge of rapidly designing a flexible SOC for low-cost networked consumer devices. UNUM uses multiple contexts to achieve zero overhead event handling and provides instructions for block data transfers that are fully integrated into the memory and cache architecture. The key advantages of UNUM are that it simplifies design of communications processors, lowers costs, closely integrates data movement and computation thereby enabling fly-by processing. It also provides flexibility by making it feasible to migrate functions to software. Our measurements show that UNUM based communications processors deliver significant performance gains for data movement compared to current general purpose processors. We also show that our software based ATM SAR implementation delivers high throughputs with low FIFO occupancy. We believe that UNUM's ability to perform fly-by processing is well suited to applications such as encryption, coding, overload control, packet classification, and packet telephony. The emergence of broadband access networks are making these applications increasingly important for low-cost consumer devices.

References

[1] *Am186CC Communications Controller User's Manual.* Sunnyvale, CA: Advanced Micro Devices.

[2] *Am79C973/Am79C975 PCnet - Fast III Single Chip 10/100 Mbps PCI Ethernet Controller with Integrated PHY Data Sheet*. Sunnyvale, CA: Advanced Micro Devices.

[3] *AMBA Specification - Rev 2.0*. Cambridge, UK: ARM.

[4] J.C. Brustoloni, P. Steenkiste, "Copy Emulation in Checksummed Multiple Packet Communication," INFOCOM97, Vol. 3, 1997, pp. 1122-1130.

[5] C. Dalton, et. al. "Afterburner," IEEE Network, July, 1993, pp. 36-43.

[6] Z. D. Dittia, G.M. Parulkar, J.R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and other Techniques," INFOCOM97, Vol. 2, 1997, pp. 823-831.

[7] P. Druschel, M.B. Abbot, M.A. Pagels, and L.L. Peterson, "Network Subsystem Design," IEEE Network Magazine, July 1993, pp. 8-17.

[8] P. Druschel and L.L. Peterson, "Fbufs: A High Bandwidth Cross Domain Transfer Facility," Proc. 14th Symposium on Operating Systems Principles, 1993.

[9] P. Druschel, L.L. Peterson, and B.S. Davie, "Experiences with a High Speed Network Adaptor: A Software Perspective," Proceedings of the Conference on Communications Architectures, Protocols and Applications, 1994, pp. 2-13.

[10] T. von Eicken, A. Basu, V. Buch, and W. Vogels. "U-Net: A User level Network Interface for Parallel and Distributed Computing," Proceedings of the 15th Annual Symposium on Operating Systems Principles, Dec. 1995, pp. 40-53.

[11] *Euphony Architecture Specification - February 7, 1996*. Florham Park, NJ: AT&T.

[12] A. Harwood, "Motorola's IP Bus: The Embedded Systems Fast Lane," Embedded Processor Forum, May 1999.

[13] *HELIUM IC-000148 Preliminary Data Sheet*, Cambridge, UK: VIRATA Ltd.

[14] H. Kanakia and D. Cheriton. "The VMP Network Adaptor Board (NAB): High Performance Network Communication for Multiprocessors," Symposium on Communication Architectures and Protocols, 1988, pp. 175-187.

[15] *NET+ARM Hardware Reference Guide*, Waltham, MA: NETSilicon.

[16] *PowerQuicc: Motorola MPC860 User Manual*, Austin, TX: Motorola Corporation.

[17] P. Z. Onufryk, "Euphony: A Signal Processor for ATM," EE Times, January 20, 1997, pp. 54-80.

[18] *PLB and OPB Macro Library - First Edition*, Research Triangle Park, NC: IBM.

[19] K.K. Ramakrishnan, "Performance Considerations in Designing Network Interfaces," IEEE Journal on Selected Areas in Communications, Vol. 11, No. 2, Feb. 1993, pp. 203-219.

[20] C.A. Thekkath, T.D. Nguyen, E. Moy, E.D. Lazowska, "Implementing Network Protocols at User Level," IEEE/ACM Transactions on Networking, Vol. 1 No. 5, Oct. 1993.

[21] *TS702 Advanced Communication Controller Data Book*, Santa Clara, CA: T.sqware Inc.