ARCHITECTURAL CONSIDERATIONS FOR CPU AND NETWORK INTERFACE INTEGRATION

THE AUTHORS DESCRIBE UNUM, AN ARCHITECTURE FOR INTEGRATING COMMUNICATIONS FUNCTIONALITY INTO THE CPU. UNUM NOT ONLY SIMPLIFIES THE DESIGN OF COMMUNICATIONS PROCESSORS BUT ALSO IMPROVES THEIR PERFORMANCE AND PROVIDES THEM WITH GREATER FLEXIBILITY.

Charles D. Cranor
R. Gopalakrishnan
Peter Z. Onufryk
AT&T Labs, Research

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 a 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. Thus, designers of communications processors are continually looking for ways to speed design and verification. Rapid 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 designed.

A large body of research and experience in the design of network adaptors for workstations exists.[1-3] 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. Designers must carefully manage latency in communications processors to reduce on-chip buffering. Network interface cards (NICs) typically used in workstations plug into I/O buses that have high latencies. This forces a 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 12 Kbytes of buffering.[4] Since communications processors often contain multiple network interfaces, placing such large buffers on chip may not be possible and is certainly not cost effective.

Space requirements in workstations are not as stringent as those in the SOC environment of communications processors. This allows workstation NICs to include considerable processing power. For example, "intelligent" NICs contain on-board processors.[2] 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

0272-1732/00/$10.00 © 2000 IEEE

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 SPEClike benchmark performance. Network adaptor research has focused on reducing memory copies and host CPU processing,[5,6] both of which lead to complex interface-specific hardware that is not appropriate for communications processors.

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.

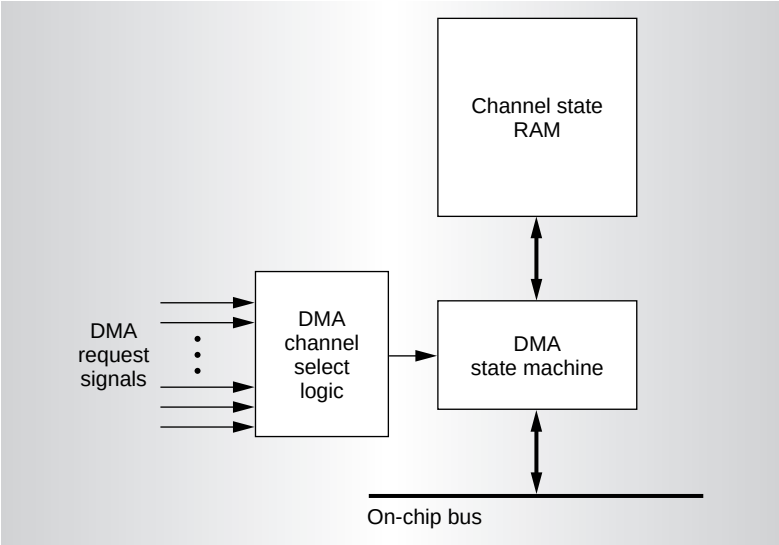

Figure 1. Multichannel DMA controller.

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 and a thriving IP 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 multichannel 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 shows a simplified block diagram of a typical multichannel DMA controller. Since only one DMA channel can be active on a bus at any given time, we can save die area 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 (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, the controller transfers its state from RAM to the DMA state machine. After the operation completes, the controller writes back the updated channel state 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. These system dependencies include 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 that is dependent on the virtual circuit iden-

tifier 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 and demultiplexing of data based on the time slot for a time-division multiplexing bus, and searching through multiple DMA descriptors for an optimal size buffer to store a received Ethernet frame. Despite the complexity of designing a multichannel DMA controller, a number of communications processors such as the AMD Am186CC,[7] the NETsilicon Net+ARM,[8] and the Euphony processor[9] use this approach.

The design of a multichannel 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 multichannel 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. The Motorola MPC860[10] and the Virata Helium[11] use this approach.

Adding a second processor for data transfers and interface-specific processing eliminates the complexity of designing a DMA and provides flexibility. However, it also introduces the software 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 processor functions must be replicated in this approach (for example, two bus interface units, two ALUs), 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.

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, an interrupt handler running on the main CPU may perform these operations. The T.sqware TS702[12] uses this approach.

We believe 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 becomes 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. UNUM is an architecture for this type of processor core.

Multithreaded CPU for event processing

Performing data movement and interface-specific processing on the same CPU as application 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 1,518-byte Ethernet frame from a receive FIFO to memory. Using a 64-byte burst transfer 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 that further increase 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. The worst-case event service latency for an interrupt with a full-context save for a high-performance embedded processor is on the order of several microseconds. Although typical performance is much better, designers must consider worst-case performance during system design since taking the best or average case will lead to conditions such as buffer overflow or underflow.

A major component of interrupt latency 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 the event service routine state from memory would still remain. This is because interrupt service routines do not retain state across invocations. For example, a data transfer event service routine must load the starting address, byte count, destination address, and possibly a descriptor pointer on entry, then update the byte count on exit.

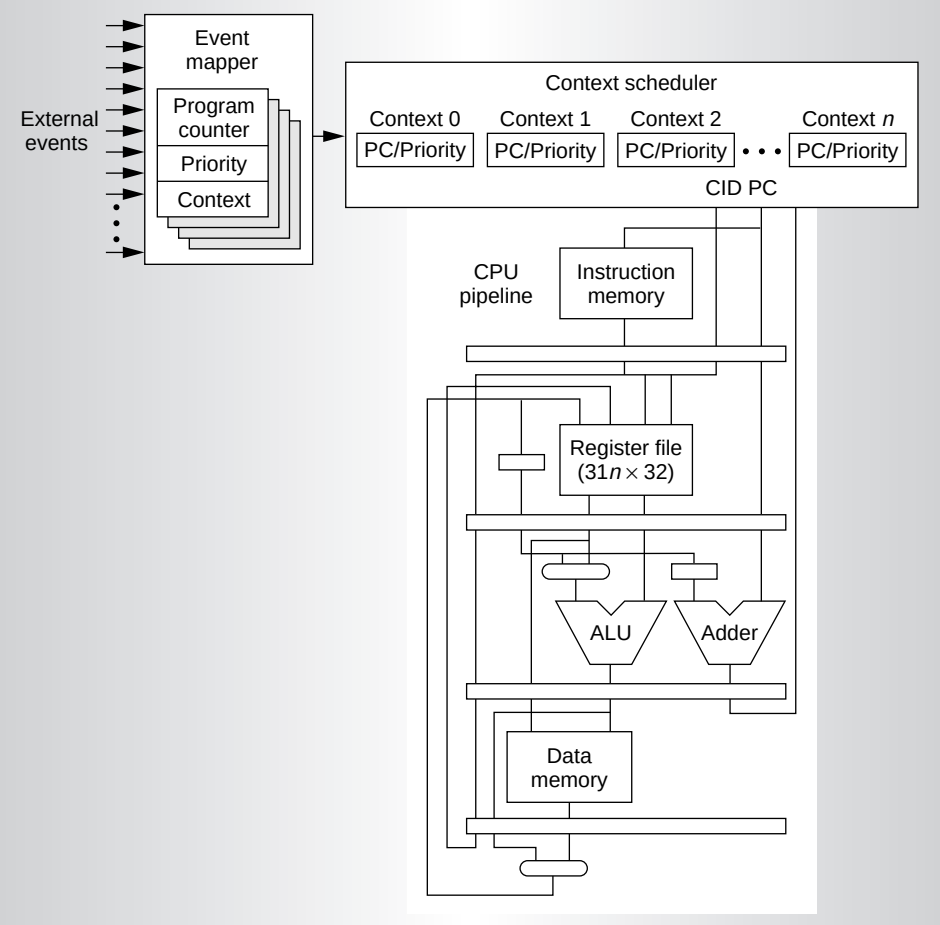

Figure 2. Example UNUM CPU.

To eliminate the latency and overhead of interrupts, UNUM employs multiple hardware contexts with priorities. By allowing the state of an event service routine to be preserved in a CPU context across invocations, we eliminate the overhead of retrieving and updating the event service routine state. Figure 2 is a block diagram of a UNUM processor. 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 when an external event occurs. 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, or multiple pending events map to the same hardware context, the event mapper uses the priority to determine the order of invocation.

The context scheduler issues 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, the scheduler issues instructions from these contexts in a round-robin manner.

The UNUM pipeline is a simple single-issue RISC pipeline augmented to support concurrent execution of instructions from multiple contexts. The 31×32 register file of a typical RISC processor is expanded to a $31n \times 32$ register file, where n is the number of supported hardware contexts. When the con-

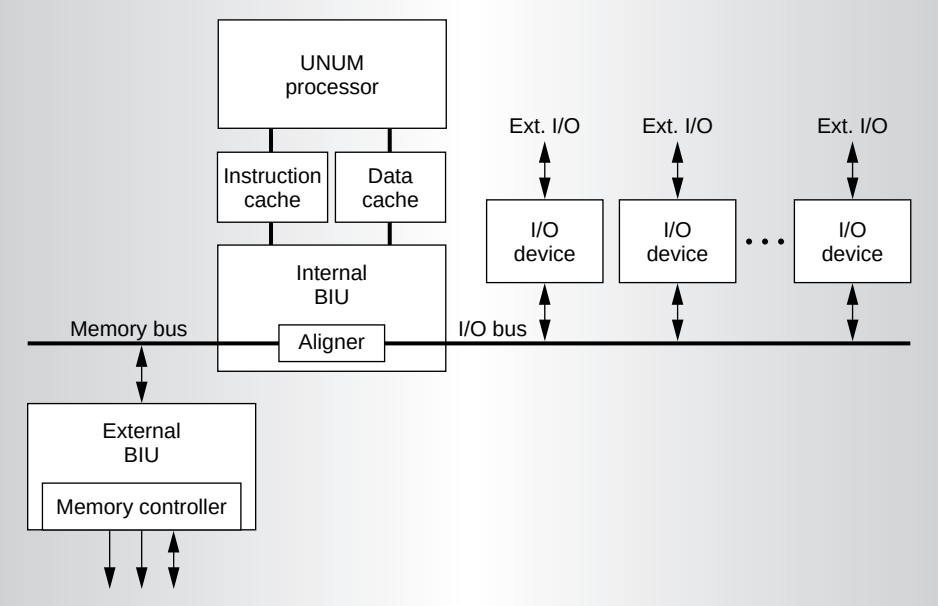

Figure 3. UNUM-based communications processor.

text scheduler selects a context from which to issue an instruction, it presents the CPU pipeline with a context ID (CID) and a program counter value. The CID, together with a register number from the fetched instruction, forms the register's actual address in the register file. Pipeline bypass and interlock logic also uses the CID. 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.

Multithreading has historically been used to tolerate memory latency. In UNUM, multithreading reduces event service latency. A UNUM processor may be designed to both tolerate memory latency and reduce event service latency.

Data movement instructions

General-purpose processors have poor data movement capabilities. Programmed I/O (PIO) generates memory-to-memory transfers that 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 noncacheable loads and stores result in single-word data transfers that achieve poor bus utilization. Using cacheable loads and stores to generate burst transfers results 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.

Since data movement is one of the primary functions of a communications processor, we have incorporated data movement instructions into UNUM. Figure 3 shows the system architecture of a communications processor based on UNUM. 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 either 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.

UNUM data movement instructions perform fly-by transfers between memory and devices on the I/O bus. The TM2D instruction transfers data from memory to an interface, while the TD2M instruction transfers data in the opposite direction. In both cases, fly-by data bypasses the data cache and does not pollute it. To maintain cache consistency, the cache supplies dirty data during TM2D processing, and performs cache invalidates during TD2M. Efficient processing of network data is supported through direct transfers between a network interface and the data cache. The TD2C instruction loads data directly into the data cache from an interface, eliminating an unnecessary transfer through memory. The TM2DD instruction discards

dirty data from the data cache as it is written to a network interface, potentially eliminating an unnecessary future write-back.

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

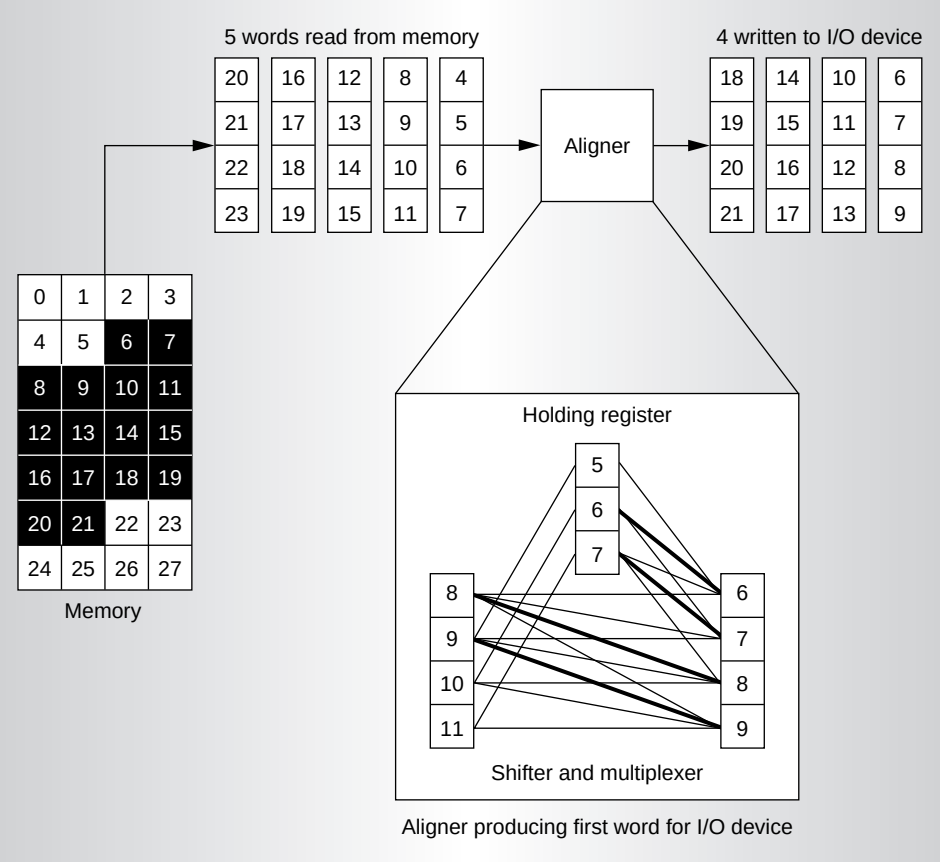

Figure 4. Unaligned 4-word fly-by transfer from memory to I/O device starting at address 6.

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. Combining a UNUM processor core with network interface cores allows communications processors to be rapidly constructed. 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 a 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.

Simulation results

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. We based the CPU in the simulator 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.

We simulated a 200-MHz UNUM processor with an 8-Kbyte, two-way set-associative instruction cache; 2-Kbyte, two-way set-associative data cache; and a 4-word write buffer. We configured our simulated 32-bit system bus to run at 100 MHz 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. Other than the ones mentioned in the next section, we did not perform hand assembly language optimizations. In addition, we assumed that event and interrupt handlers were locked in the instruction cache.

Data movement

For our initial measurements we wrote a data movement micro-benchmark that simulated the transfer of a 1,518-byte packet from

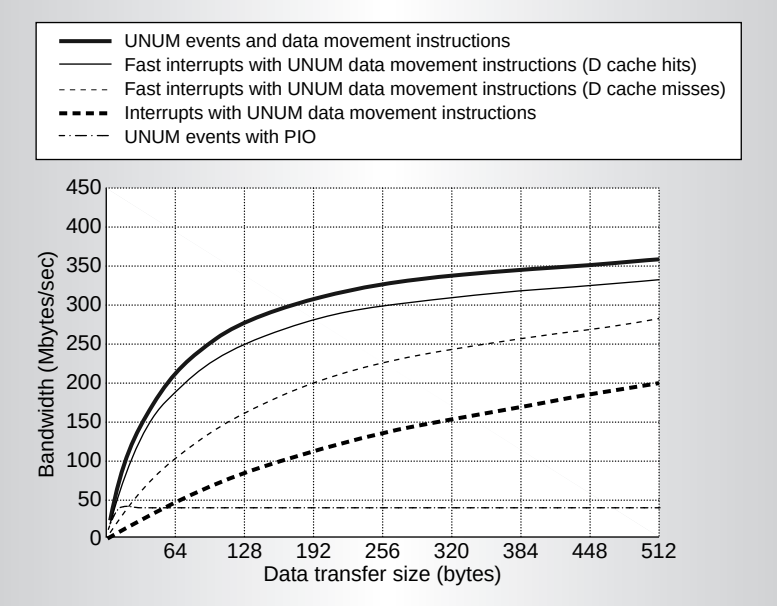

Figure 5. Data movement performance.

memory to a network interface using a range of burst transfer sizes. We 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 1 µs. 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 provides the results of the data movement benchmark. PIO-based data movement results in the worst performance. The highest achievable bandwidth using PIO was 40 Mbytes/sec. This was true regardless of the type of CPU used (UNUM, fast interrupts, or regular interrupts) and caching assumptions since 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 Mbytes/sec. Data cache misses had little effect since we assumed a fixed interrupt overhead of 1 µs, which dominated. Replacing normal interrupts with fast interrupts improved performance to 104 Mbytes/sec, assuming all data cache misses, and 189 Mbytes/sec, 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 Mbytes/sec. 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.

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.

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 in which the processing performed on received data depends 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 that indicate the data length, a user-to-user byte, and a CRC-32 value. SAR receive processing consists of reassembling received cells 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 identifier field (VPI/VCI) in the cell header is used to

look up the virtual circuit that the cell belongs to. This lookup returns a data structure that 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 (for example, 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. We wrote code for 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.

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 750 ns.

Our second experiment measured UNUM processor utilization and the 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 1,536 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, which determines required on-chip buffering. Using UNUM, a 25-Mbps, full-duplex line rate requires just a four-cell transmit and receive FIFO.

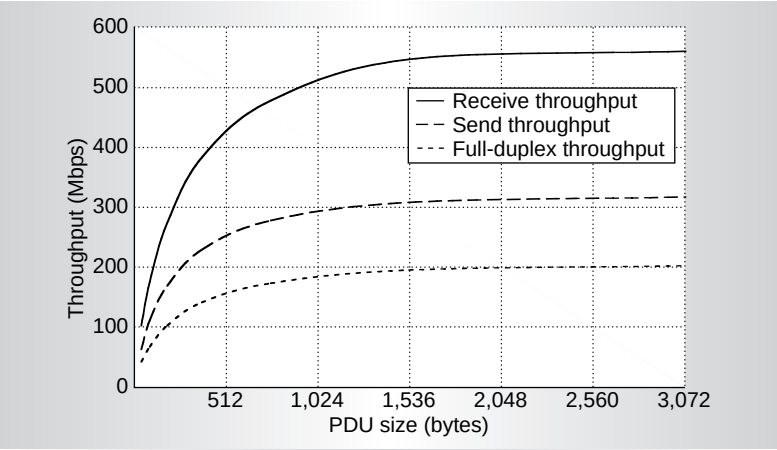

Figure 6. Maximum ATM SAR throughput.

UNUM simplifies design of communications processors, lowers their cost, and closely integrates data movement and computation, thereby enabling fly-by processing. 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 is making these applications increasingly important for low-cost consumer devices. We are continuing our investigation of UNUM for these and other applications areas. MICRO

References

1. C. Dalton, et. al., "Afterburner," *IEEE Network,* Jul. 1993, pp. 36-43.
2. H. Kanakia and D. Cheriton, "The VMP Network Adaptor Board (NAB): High Performance Network Communication for Multiprocessors," *Proc. Symp. Communication Architectures and Protocols,* ACM, New York, 1988, pp. 175-187.
3. K.K. Ramakrishnan, "Performance Considerations in Designing Network Interfaces," *IEEE J. Selected Areas in Communications,* Vol. 11, No. 2, Feb. 1993, pp. 203-219.
4. *Am79C973/Am79C975 PCnet—Fast III Single Chip 10/100 Mbps PCI Ethernet Controller with Integrated PHY Data Sheet,*

Advanced Micro Devices, Sunnyvale, Calif.
5. P. Druschel et al., "Network Subsystem Design," *IEEE Network,* Jul. 1993, pp. 8-17.
6. T. von Eicken et al., "U-Net: A User Level Network Interface for Parallel and Distributed Computing," *Proc. 15th Ann. ACM Symp. Operating Systems Principles,* ACM, Dec. 1995, pp. 40-53.
7. *Am186CC Communications Controller User's Manual,* Advanced Micro Devices, Sunnyvale, Calif.
8. *NET+ARM Hardware Reference Guide,* NETsilicon, Waltham, Mass.
9. P.Z. Onufryk, "Euphony: A Signal Processor for ATM," *EE Times,* Jan. 20, 1997, pp. 54, 80.
10. *PowerQuicc: Motorola MPC860 User Manual,* Motorola, Austin, Tex.
11. *HELIUM IC-000148 Preliminary Data Sheet,* VIRATA, Cambridge, UK.
12. *TS702 Advanced Communication Controller Data Book,* T.sqware Inc., Santa Clara, Calif.

Charles D. Cranor is a senior technical staff member at AT&T Labs–Research in Florham Park, New Jersey. His interests include networking, operating systems, and computer architecture. Cranor received a bachelor's degree in electrical engineering from the University of Delaware and a master's and doctorate in computer science from Washington University in St. Louis, Missouri. He is a member of the IEEE, ACM, and USENIX, and a kernel developer for the open-source BSD operating systems projects.

R. Gopalakrishnan is a senior technical staff member at the AT&T Labs–Research in Florham Park, New Jersey. His interests include packet telephony systems, service differentiation and I/O performance optimizations in server operating systems, multimedia networking, and IP multicast. Gopalakrishnan received the BTech degree in electrical engineering from IIT Kanpur in India, the MTech degree in computer science from IIT Delhi, and the DSc degree in computer science from Washington University, St. Louis. He is a member of the ACM.

Peter Z. Onufryk is a technology consultant with AT&T Labs–Research in Florham Park, New Jersey. He was the lead architect of two communications processors and has worked on a number of research and military computers. Onufryk received his bachelor's degree in electrical engineering from Rutgers University, master's in electrical engineering from Purdue University, and PhD in electrical and computer engineering from Rutgers University. He is a member of the IEEE, IEEE Computer Society, and ACM.

Direct comments about this article to P.Z. Onufryk, Room B009, AT&T Labs–Research, 180 Park Ave., Bldg. 103, Florham Park, NJ 07932; pzo@research.att.com.