Implementing Application-Specific Cache-Coherence Protocols in Configurable Hardware

David Brooks and Margaret Martonosi
Dept. of Electrical Engineering
Princeton University
{dbrooks,mrm}@ee.princeton.edu

Abstract

Streamlining communication is key to achieving good performance in shared-memory parallel programs. While full hardware support for cache coherence generally offers the best performance, not all parallel machines provide it. Instead, software layers using Shared Virtual Memory (SVM) can be built to enforce coherence at a higher level. In prior work, researchers have studied application-specific cache coherence protocols implemented either in SVM systems or as handlers run by programmable protocol processors. Since the protocols are specialized to the needs of a single application, they can be particularly helpful in reducing the long latencies and processing overhead that sometimes degrade performance in SVM systems.

This paper studies implementing application-specific protocols in hardware, but not via an instruction-based protocol processor as is typical. Instead, we consider configurable implementations based on Field-Programmable Gate Arrays (FPGAs). This approach can be faster than software-based techniques and less expensive than some hardware-based techniques. We study one application, appbt, in detail, including a VHDL-level design of the configurable protocol design. We sketch out approaches for other applications as well. Implementing protocol operations in configurable hardware improves communication performance by roughly 11X for a 32-node system. While overall speedups are a more modest 12%, our method is still promising because of its flexibility and because it offers a new way of harnessing configurable hardware at the network interface, where it already exists or could be easily added to current systems.

1. Introduction

Writing shared-memory parallel programs is thought to be easier than message-passing programs because of the simplified memory and communication model involved. Supporting fully cache-coherent shared-memory in hardware, however, can be expensive. Some systems instead opt to implement a shared-memory programming model using a software-based shared virtual memory (SVM) system [1].

Whether implemented in hardware or software, the key to good shared memory performance lies in the protocol implemented. To address this, prior research has considered implementing application-specific protocols. In such approaches, the cache coherence protocol is specialized to the communication needs of a particular program. Such protocols are possible in cases where the coherence mechanism (either hardware or software) can be changed or customized at program run-time. Past work has evaluated such protocols running in SVM software on the main compute nodes themselves, or in handler code running on separate protocol co-processors.

Our work investigates a third option: implementing application specific protocols using a “configurable” hardware approach based on Field-Programmable Gate Arrays (FPGAs). These SRAM-based chips can be infinitely reprogrammed just by downloading a new stream of bits to rewrite configuration settings. Once configured, they behave like hardware however, with a gate-based, rather than instruction-based interface to their functionality. Since current network interface boards like Myrinet already contain FPGAs (for other purposes) it seems natural to evaluate their utility for application-specific protocols. Only small changes to existing network interface boards would be needed to make the proposed ideas feasible.

Studying shared memory approaches and prior research in application-specific protocols we note:

1) Flexible protocols can be conveniently implemented in configurable hardware, rather than in software. This facilitates overlapping computation and communication and can also accelerate the protocol handlers themselves.
2) Coherence protocols have characteristics amenable to FPGA computing: bit manipulation, hardware parallelism, and simple integer computations.
3) Existing tools designed to facilitate developing application specific protocols in software can be retargeted to automatically synthesize hardware implementations.
4) The network interface boards that interconnect compute nodes typically have several FPGAs on them anyway. Only minimal industry cooperation would be needed to get a bit more space on them for implementing protocols in them.

With these observations in mind, this paper explores the possibility of implementing an application-specific protocol processor in configurable hardware. A detailed study for one application, *appbt*, showed an 11x speedup in communication time compared to other implementations. Other applications show viability as well.

Sections 2 and 3 discuss previous research efforts to implement application-specific coherence protocols and why we believe that configurable hardware is a viable alternative. Section 4 gives an in-depth description of our proposed architecture. In Section 5, we outline the methodology that we use to evaluate our architecture. In Section 6 we evaluate the feasibility and promise of this new method with a detailed case study using *appbt*. Section 7 investigates additional parallel applications with general descriptions of possible implementations. Section 8 discusses potential difficulties and future work. Section 9 presents our conclusions.

2. Why Application-Specific Protocols?

Recently, application specific protocols have been recognized as a valid means of improving protocol performance. Several different strategies have been proposed for their implementation. One approach to the implementation of application-specific coherence protocols has been the development of *Tempest* by the Wisconsin Wind Tunnel Project [2]. *Tempest* aims to provide a

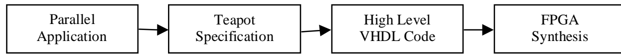

Figure 1 – Proposed design flow.

standard, system-independent parallel programming user/system interface that offers programmers access to a variety of different communication mechanisms, including active messages, bulk data transfer, virtual memory management, and fine-grain access control [3].

Tempest defines the architecture of a communication interface for shared-memory parallel programs; *Blizzard* is one implementation of that architecture [4]. *Blizzard* runs the coherence protocol code in software on each of the main compute nodes. Studies with *Blizzard* on applications with a wide variety of communication patterns have shown that application-specific coherence protocols can provide substantial speedups over even carefully-tuned implementations using a stock coherence protocol. This approach allows great flexibility in customizing the protocol to application characteristics as everything is done in software. Naturally a major disadvantage is that it slows down the host processor since it is responsible for both computation and protocol processing. Another speed disadvantage is that the host processor is not physically located next to network interface and an associated DMA engine.

These disadvantages have spurred an interest in exploring moving functionality from the main compute nodes down into the network interface. In some studies, such functionality is implemented as extra handler code run by a programmable network interface processor such as the LANai processor in a Myrinet network interface [5] [6] [7]. Other approaches have provided even more aggressive levels of hardware support, up to full hardware cache coherence [8] [9]. Our proposal, which implements protocol processing in configurable FPGA chips on the network interface, represents an intermediate position between full-hardware or full-software implementations.

3. Why Configurable Hardware?

Field Programmable Gate Arrays (FPGAs) allow the hardware functionality of a chip to be infinitely reprogrammed through a stream of configuration bits. Thus, unlike an EPROM, an FPGA can be reprogrammed simply by downloading new configurations to its SRAM-based configuration memory bits. Because FPGAs are fabricated with the same manufacturing process as CMOS SRAMs, they can be low-cost commodity parts. The inexpensive hardware flexibility of FPGAs has led to their use in areas traditionally associated with custom hardware. This has been especially true for rapid prototyping and low-volume production. More recently researchers have attempted to find ways to utilize the ability of FPGAs to be reconfigured within systems. Our research demonstrates the use of configurable hardware to implement application-specific coherence protocols within SVM systems.

Reconfigurable hardware has several unique features that are amenable to protocol processing. Since the protocol processing hardware is customized for each application, all of the available resources can be used for the particular application. In addition, configurable hardware inherently allows extensive fine-grain parallelism and in no way restricts the designer to sequential execution. Finally, FPGAs are well suited to the types of computation prevalent in protocol processing: integer-oriented address calculations, counter operations, and bit-manipulations.

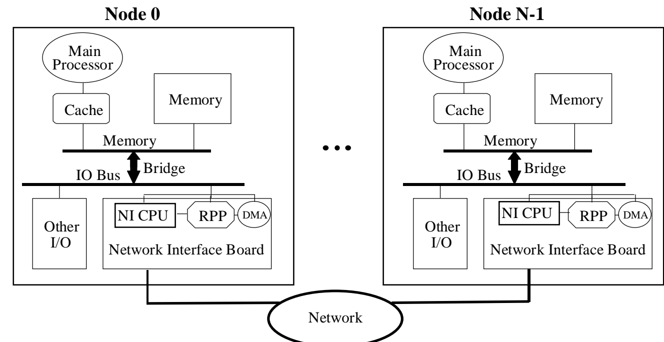

Figure 2 – Proposed system architecture.

One obstacle to the acceptance of an architecture like the RPP is that writing application-specific coherence protocols in software can already be a challenge; implementing hardware designs seems even tougher. Researchers at Wisconsin have developed a language called Teapot which aids programmers in writing and verifying coherency protocols [10]. We can circumvent the application-specific hardware hurdle by implementing a VHDL backend for Teapot. This would allow the automatic synthesis of FPGA-based hardware from a Teapot specification.

Figure 1 outlines a potential design flow for the design of an FPGA-based protocol processor. First, the sharing patterns of the parallel application must be analyzed and described in a language such as Teapot. The Teapot compiler would then create high level VHDL code to be passed to commercial CAD tools for synthesis into the FPGA configuration bitstream.

4. Our Proposal: A Reconfigurable Protocol Processor

Figure 2 shows a diagram of the proposed system architecture. The reconfigurable protocol processor (RPP) is tightly coupled to both a DMA engine and the network interface (NI) CPU. This is similar to the Myrinet network interface. It allows the protocol processor to closely interact with the DMA and the NI with FIFOs serving as buffers between parts. Another advantage to this architecture is that FPGAs are already available on some current network interface boards [5]. Thus realistic implementations of similar architectures are quite feasible in the near-term.

The proposed reconfigurable protocol processor system offers a wide range of performance benefits for providing application-specific protocols:

Background protocol processing: Software SVM rely on the microprocessor for protocol processing; they must stall main program execution and incur interrupt overhead in two cases: (i) whenever a message is prepared and sent to the network interface and (ii) whenever an incoming message is received at the network interface. With the RPP system, the extra hardware can send or receive a message or other protocol event, process the event, and transact with main memory, leaving the microprocessor to continue with program computation.

Fast, Intelligent DMA: In our proposed architecture, the RPP is closely coupled to a DMA engine. There are two major reasons why this is beneficial. First, we achieve the benefits of fast,

intelligent data transfer from the network interface directly to memory, or vice-versa. Allowing the RPP, rather than the compute node, to control the DMA reduces the performance degradation when sending short, non-contiguous sections of memory, because it can easily be customized for strided accesses.

Specialized processing on both sending and receiving messages: In managing communication, the compute node is no longer limited to simple, general-purpose protocol commands such as "Send memory location 17 to node 2," but can issue brief, application-specific commands such as "Send update data pattern 8". The RPP interprets these and expands them into a complicated message. Extremely brief commands by the microprocessor can set the RPP at work doing complex processing. This improves communication/computation overlap and also reduces the software overhead of communication processing. Likewise, on the receiving end of messages, entire message handlers can be implemented by the RPP, keeping all of the message processing away from the microprocessor.

5. Methodology

In order to evaluate the proposed reconfigurable protocol processor, we devised a simulation environment that allows us to realistically compare applications running on an RPP system to those running on a software SVM system. Several simulation models were developed to achieve this, as described below. First, VHDL designs and simulations were used to verify our design and to determine feasible clock speeds. Our second model simulates the performance of the entire system when a particular application runs with its RPP configured for that application. Finally we simulate the performance of the system when an application uses software-based coherence.

5.1 VHDL RPP Model

In the final system, a Teapot-VHDL translator would facilitate RPP design. Here, however, we hand-designed the RPP in order to get performance estimates for it. To simulate RPP performance for a given application, the RPP is first designed at a register-transfer level, and then state transition diagrams are constructed to determine how many cycles various protocol functions will take to execute. Finally, a full VHDL design determines the cycle time for the RPP. Section 6.3 elaborates on the VHDL design for the detailed *appbt* case study.

5.2 Multiprocessor Application Simulator

The high-level application simulator is based on MINT, a multiprocessor, event-driven simulator [11]. MINT simulators accept an application program as input and simulate the program's performance, using a user-defined back-end. MINT passes all read, write, and other relevant events to the back-end. Here, we simulate the operation of coherence protocols as they service memory requests and maintain coherency. In addition, user-generated events simulate other functionality, such as bulk-data transfers or protocol-specified active messages.

5.3 System Timing

5.3.1 Protocol Timing

We consider three types of costs in simulating the message handlers for the SVM system:

- A handler dispatch of 150 processor cycles to save the machine state and start an interrupt.
- The handler functions incur latency to start each transfer or receipt of data from the network interface, and take time to actually transfer data over the PCI bus.
- The processor takes a small amount of time to perform synchronization operations within message handlers – for instance, the clearing or setting of a counter. As a low estimate, we charged the message handler 1 cycle for each counter clear and 2 cycles for each counter increment.

5.3.2 Baseline Compute Node Timing

For our baseline system, we assume a microprocessor running the MIPS instruction set at 300 MHz. The processor has an on-chip write-through data cache with 32-byte lines and 1024 entries. Cache misses take 20 processor cycles to fill, using the memory bus. The baseline configuration utilizes 32 processors in a grid interconnect network.

We also assume a 33 MHz, 64-bit PCI bus serving as the I/O bus for each processing node [12]. This is consistent with the speeds of current Myrinet network interface boards. We assume that each PCI transaction is limited to at most 64 PCI cycles. Reads or writes to sequential addresses need no time between sending or receiving 64-bits of data. Between reads to non-consecutive addresses, a turn-around cycle is required between the cycle where the master drives the address on the bus and the cycle where the memory responds with data. Between writes to non-consecutive addresses, no turn-around cycle is necessary because we are able to make use of the PCI "Fast Back-to-Back Transactions" functionality [13].

5.3.3 Network Timing

Our network timing is modeled after the Myricom networks [5]. To simulate wormhole routing, we assume that:

Message Delivery Time = (Number of Hops) * Cycles per Hop + (Message Size –1) * NetStep

The NetStep we assume to be 20ns, and the cycles per hop is set at 100ns. Message size is measured in 16-bit chunks. To limit simulation time, we simulate a contention-free network because previous work has shown that network contention is not a major bottleneck in these applications [14]. We also assume a device driver overhead of 40 processor cycles for each transmission or receipt of data from the microprocessor to the network interface or the RPP.

6. A case study: *Appbt*

To demonstrate our idea, we have performed a detailed design and evaluation for an RPP customized for the *appbt* program. *Appbt*, one of the NAS parallel benchmarks, is an iterative, three-dimensional, computational fluid dynamics application [15]. For each iteration, it performs a number of calculations to compute new values for each grid point. Some calculations rely on values for that grid point only, while other calculations also utilize values that are one or two grid points away. The "value" of each grid point consists of 90 different double precision floating point

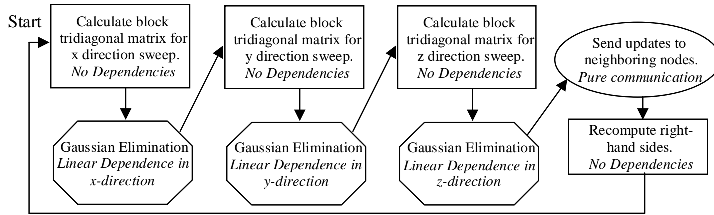

Figure 3 – Overall structure of an Appbt iteration, showing computation and communication. This structure describes all iterations after the first iteration.

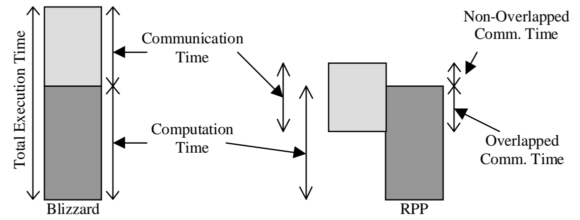

Figure 4 – Communication in Blizzard vs. RPP.

numbers that summarize the state of the fluid being studied. The Wisconsin Wind Tunnel (WWT) project has optimized *appbt* in order to measure the potential speedups from application-specific coherence protocols on a parallel shared memory system. We used their version of *appbt* as a starting point for our research.

Speedups from our reconfigurable protocol processor will only affect the communications aspects of program execution time, because we are only focusing on improving the communication handlers. Thus, we next describe these communication patterns in detail.

6.1 Appbt Communication

Figure 3 shows a flowchart of the communication that occurs in *appbt*. There are two major types of communication – communication during the Gaussian elimination phase and communication during the update phase.

Gaussian elimination: Gaussian elimination phases occur three times per iteration to transmit newly calculated values in the x, y, and z directions for both "forward elimination" and "back-substitution." Forward elimination transmits newly calculated values to dependent grid points on the right. Back substitution transmits values to the left. The RPP has separate communication handlers for each of these two types of communication.

Update Phase: The update phase of communication occurs shortly before the end of each iteration. Here, each node sends updates to all neighboring nodes. As shown in Figure 3, the updates are the entire face, two grid points deep, shared between a node and its neighbor. Since these faces are not needed until the beginning of the next iteration, we hide some inter-node communication delay behind the remaining computation time. Again we have implemented separate communication handlers for each update axis (x, y, and z).

6.2 *Appbt* communication speedup methods

In this section we describe the enhancements that our reconfigurable protocol processor incorporates in order to improve communication time in *appbt*. In the original *Blizzard* systems, all protocol handling is performed by the compute node. Thus, there is no way to overlap protocol handling with compute time. As shown in Figure 4, the RPP decreases communication time in two main ways:

- Decreasing the overall amount of communication time.
- Overlapping some communication with computation.

Note the computation time stays constant with both systems. Only the communication time is decreased and overlapped with computation.

In the following paragraphs we explore in detail the enhancements that the RPP uses to decrease the communication time. The first three methods require the additional hardware that the RPP provides. The last two techniques could be applied to a software implementation but are more convenient with the RPP.

Background message processing: In some cases, messages sent to a processor may not be needed immediately. For example, update messages are not needed until the beginning of the next iteration. In such cases, computation may continue before all the messages have arrived. With the software implementation, the processor stops computation and handles an interrupt for each message that arrives. With the RPP implementation, the processor never has to process an intermediate interrupt to service a message. Rather, the RPP deals directly with memory, placing the arriving updates in their appropriate memory locations. A counter, discussed below, notifies the processor when all updates have arrived.

Fast, Intelligent DMA: As previously discussed, the close association of the RPP with the DMA engine allows intelligent, application-specific DMA transfers, particularly smaller, non-contiguous transfers. In *appbt*, combining smaller messages into larger ones involves accessing multiple non-contiguous sections of memory. With the RPP, such non-contiguous memory accesses are simple. The RPP splits messages into their component parts, does some simple address calculations while waiting for the current DMA operation to complete, and feeds data and addresses to the DMA engine until the message is complete. Using the RPP also decreases message size (since the receiving RPP can also calculate much address and length information on its own).

Synchronization Squashing: Regardless of whether the update messages are sent as single grid points or entire faces, any given node will expect multiple messages from multiple processors before it can proceed with more computation. In software, synchronization for this is implemented with a software counter for the x, y, and z directions. The data received is unused until the counter indicates that all the data for that dimension has been received.

With the RPP much of this processing can be avoided. When the program begins, the microprocessor sends a message to the RPP telling it how many updates to expect from each dimension. Since the communication is totally static, this can be fixed through the entire execution. Whenever the RPP receives an update message, it sends the update data to memory. But instead of incrementing a software counter, the RPP simply increments its own version of the counter located on chip. When the counter

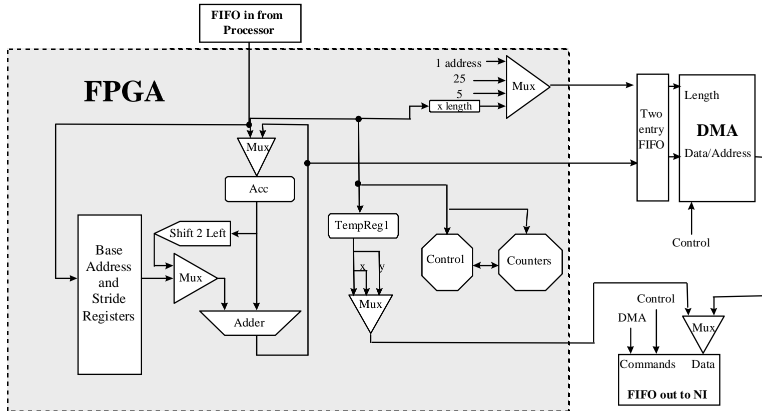

Figure 5 – Block diagram of Send datapath.

	Cycle Time	Occupied CLBs	F&G Function Gens.
Send	61.4ns (16.3Mhz)	558	835
Receive	60.1ns (16.6Mhz)	496	757

Table 1 – Cycle times from *xdelay* with 4013EHQ240-2.

reaches the critical value, the RPP knows that all updates for that dimension have been received, so it writes the counter value to main memory. Thus, many intermediate reads and writes to increment the counter are reduced to a single write when the data is ready.

Forward Elimination: In a software-based implementation, each data transmission during the forward elimination stage actually consists of 11 different messages, each 5 doubles long. With the RPP, however, all the data may be sent as one message that the receiving RPP splits to place in the appropriate areas of memory. This is possible because the sending and receiving RPPs can perform address calculations and control the transfer of data from memory to the NI and vice-versa. Thus, 11 messages are reduced to one longer message.

Update: During the end-of-iteration update messages, the *Blizzard* implementation forwards each grid point as separate message of 5 doubles. Again the RPP can be configured to accept a much longer message, break it into its component parts, and write the appropriate data to memory. In our implementation, we chose to have the RPP send the data in whole-face chunks, so that an update to a neighboring processor takes two messages.

6.3 VHDL Design of RPP

To get accurate results with our simulator, a full VHDL implementation of the RPP was necessary in order to determine the speed of our design which when tailored for *appbt*. The hardware cost of two Xilinx 4013 FPGAs is also minimal compared to that of a custom ASIC. The design was originally intended to occupy one FPGA. A configurable logic block (CLB) is the basic unit of which FPGAs are comprised. It can implement roughly 10 simple gates of logic and includes 2 flip-flops for state.

In order to meet pin and CLB constraints, we made a decision to split the design into one FPGA for sending and one FPGA for receiving. However, with the more advanced FPGA technology available today, the entire design would easily fit onto one chip. Figure 5 shows a block diagram of the send datapath that we used for *appbt*. The control logic consists of the finite state machines for the five send handlers and five receive handlers. The simplest handlers consist of eight sequential state transitions, while the more complex handlers have a few more states with loops. The receive datapath is very similar and much of the logic could be shared. However, the control logic made the design too large to fit onto the Xilinx 4020 FPGA. A 40-bit version (capable of 40-bit address calculations) was implemented in VHDL and synthesized using FPGA Express [16]. ViewSim was used for simulation [17]. Xilinx place and route tools were then used to generate bitstreams and determine cycle times [18]. See Table 1 for results.

6.4 Simulation Results

Since the computational core of the application remains the same, all speedups result from reducing communication time or from overlapping it with computation. In the first subsection we explore in detail the results from a baseline system. In the following subsections we investigate the effect of varying the system parameters.

6.4.1 Baseline System

We assume a base system with 32 processing nodes each containing one 300 MHz microprocessor running the MIPS instruction set. We also assume a 33 MHz, 64-bit PCI bus as the I/O bus in each processing node. The default problem size is a 12x12x12 array with 60 iterations.

For this baseline system, we achieved a communication time speedup of 10.84 for the RPP system compared to *Blizzard*. In terms of the definitions introduced in Figure 4, this means that the non-overlapped communication time with the RPP system is nearly 11 times smaller than the time *Blizzard* spent on communication. Part of the reason for this speedup is that a large portion of the communication time was overlapped with computation. Overlapping communication with computation is a key technique that coprocessors like the RPP use to improve runtimes. For *appbt*, the amount of overlap varies by processor. Processors in the center of the network have more update messages. Update messages have more potential for overlap because there is additional computation time available to hide communication delays while the program is in the "Re-compute right-hand sides" phase (see Figure 3). For the baseline system, 45.7% of the communication was overlapped on average. 52.6% of the total communication time for update messages were overlapped while 28.5% of the total communication time for Gaussian elimination messages were overlapped.

Because all of our speedup comes via improved communication, the fraction of overall execution time that *appbt* spends in communication is critical. With a software version, 11.6% of the overall time is spent in non-overlapped inter-node

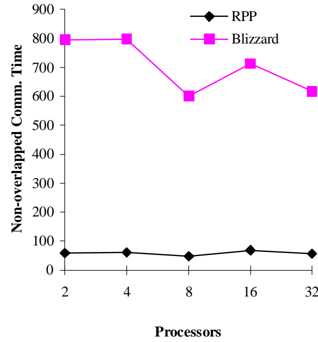

Figure 6 – Non-overlapped communication time for RPP and Blizzard (avg num. Kcycles per iteration).

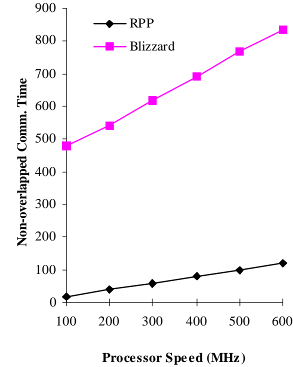

Figure 7 – Non-overlapped communication time for RPP and Blizzard (avg num. Kcycles per iteration).

communication. The RPP's enhancements bring this number down to 1.2%. This reduction in communication time gives the RPP system a speedup of 1.12X over a *Blizzard* system for overall program execution time. In comparison, an ideal system that assumes all communication occurs instantaneously, would achieve a speedup of 1.13X over *Blizzard.*

While a total execution time improvement of 12% is not phenomenal, it is important to note that *appbt* is a full benchmark, not just a kernel with heavy communication. In this paper we focus on the communication aspects of the application – the portion of the application that the RPP succeeds in speeding up significantly – while keeping in perspective that the overall execution time is important as well.

6.4.2 RPP Speedups for varying numbers of processors

In our baseline system, we assume 32 processors, but smaller configurations are also common for low-end parallel processing systems for which the RPP might targeted. Here, we compare the *Blizzard* and RPP protocol processing methods on a common *appbt* input set while varying the number of processors in the system. For the total execution time, we compare the RPP's speedup over *Blizzard* to that of an ideal protocol processor with instantaneous message transmission.

Figure 6 shows the non-overlapped communication time in *appbt* for *Blizzard* and the RPP. From this figure we can see that for 32 processors there is a communication time speedup of 10.8. As we move down to 2 processors, the speedup increases to 13.5, a change of 25%. The fluctuations in the communication time are due to the way that each processor's subcube is partitioned. The size and shape vary as we change the number of processors, and this causes significant changes in the communication patterns. These increases in speedup correlate to the amount of communication that can be overlapped. For 32 processors, the overlap averages 45.7%. As we move down to 8, 4, and 2 processor networks the percentage of overlapped communication rises into the 70s.

For the overall execution time speedup the amount of time that is spent in communication is critical, because this is the portion of the overall program execution that is reduced by protocol operations. As we increase the number of processors in the system, the amount of communication increases. For example at 2 processors Blizzard spends 1.5% of its total execution time in communication, but at 32 processors this number rises to 11.6%. Thus, as we move to systems with more processors, the RPP systems shows greater overall execution time speedups. For 2 processors, the execution time speedup was just under 1.02X, but at 32 processors this increases to 1.12X.

6.4.3 RPP Speedups for varying microprocessor speeds

Our baseline system, which assumed 300MHz, is about the clock rate of many current-generation microprocessors. Investigating the effects of higher microprocessor speeds is interesting as we look to the future. It is also interesting to explore the speedup that the RPP would obtain on lower-end microprocessors. The RPP's low hardware and design costs make it well suited for low cost multiprocessors which would tend to have lower end microprocessors. We compared the *Blizzard* and RPP protocol processing methods on a common *appbt* input set while varying the speed of the host processors. For the total execution time, we again compare the RPP's speedup over *Blizzard* to that of an ideal protocol processor with instantaneous message transmission.

Figure 7 shows *appbt*'s non-overlapped communication time for both systems. Both times increase as the microprocessors become faster. Since we have assumed the RPP cycle time to remain constant while a software approach benefits from clock rate improvements, the RPP is slower relative to *Blizzard* at high clock speeds. We would expect, however, that future generations of FPGAs would allow the RPP to increase in performance with the microprocessor. Furthermore, limitations of the 33MHz PCI bus become a factor at the high clock speeds. For these reasons our communication speedup varies considerably – from 25.2 for 100MHz processors to 6.9 for 600MHz processors.

The overall execution time speedup vs. *Blizzard* increases as we move to higher clock rates. For example, from 100MHz to 600MHz the speedup increases from 1.09 to 1.15. This is because the faster processors decrease the amount of time spent on computation. Hence, communication plays a larger role in the overall program execution time at higher clock speeds and this benefits the total execution time speedup. For *Blizzard's* 100MHz system, the fraction of time in communication was 9.4%. This increases to 14.8% for the 600MHz system. For the RPP, these numbers drop to 0.4% and 2.4% respectively.

7. Other Applications

The main focus of our discussion so far has been on the parallel application, *appbt*. The RPP techniques that were

successful in improving *appbt*'s execution time are also applicable to a wide range of applications. Without going to the same level of detail, this section discusses RPP implementations for other applications, keeping in mind the strategies discussed in Section 4:

- Background protocol processing
- Fast, Intelligent DMA
- Specialized processing on both sending and receiving messages

7.1 *EM3D*

EM3D is another parallel benchmark application that models the propagation of electromagnetic waves through objects in three dimensions [19]. The program contains a set of E nodes, which represent electric fields, and H nodes, which represent magnetic fields. E and H nodes are arranged in a bipartite graph with directed edges linking E and H nodes that depend on each other. During each iteration, E nodes are updated based on the weighted values of neighboring H nodes and vice-versa. A common parallel implementation divides regions of E and H nodes up into computing node segments. Communication between processors occurs when E and H nodes are connected by a "remote edge", meaning that the neighboring E and H nodes are on different processors.

During the first iteration, the communication pattern has not been established so a general purpose protocol must be used. After the first iteration of *EM3D*, the sharing pattern between processors has been set and will stay the same for all subsequent iterations. That is, when an E or H node is updated, there is a known list of compute nodes that need to be sent a message with the updated data. At this point, application-specific update protocols can take over to enhance communication efficiency. The application-specific protocol for *EM3D* updates all remote nodes at the end of each half-iteration [4]. This approach made sense there because the processor is busy calculating new values of the current nodes, so it has no free CPU time available for protocol processing. In addition, leaving all of the updates to the end of the half-iteration allows one large message to be sent to each processor that needs update data.

Now we consider how an RPP could be used to decrease communication overhead. With an RPP most protocol processing could be entirely overlapped with computation. After each node has been calculated, the RPP sends that update message to the dependent processing node. The closely-coupled DMA engine allows efficient transfer of these small messages. Groups of messages are packaged and sent to a specific processor as in *appbt*. Furthermore, the RPP performs all protocol synchronization support which includes deciding when all incoming dependent nodes have been received. This type of synchronization is very efficient in FPGAs with bit manipulations.

Chandra, Larus, and Rogers have provided an in-depth analysis of communication and computation time for shared-memory and message-passing versions of *EM3D* [20]. The analysis notes that a *Blizzard* implementation running application-specific protocols allows the shared-memory version to perform equivalently with the message-passing version. Using this conclusion we are able to roughly estimate the amount of speedup that the RPP could achieve. For the main loop there would be 26.5M cycles of computation and 40M cycles of communication. Thus approximately 26.5M cycles of communication would be overlapped with computation. The remaining 13.5M cycles of communication would likely be reduced by RPP enhancements. However, as a low-estimate to potential speedup we assume that these non-overlapped communication cycles remain the same. This results in a total execution time improvement of about 51% for the RPP over *Blizzard*. The increased amount of time spent in communication explains why *EM3D* promises more improvement on the RPP than *appbt*.

7.2 *Unstructured*

Unstructured is based on a computational fluid dynamics application that uses an unstructured mesh to model a physical structure [21]. Nodes make up the structure of the mesh and are connected by edges, when in pairs, and by faces, when in groups of three or four. A common parallel implementation groups related nodes together and then partitions edges onto various processors. Computation involves iterative loops over nodes, edges, and faces, and thus edges and faces that span processors will require shared data to maintain coherency. Like *EM3D*, the communication dependency pattern is fixed after the first iteration.

Because of the extensive amount of time spent in communication, *unstructured* is well-suited to an RPP implementation. Inter-node dependency synchronization could be taken care of by the RPP for each node in the system. The RPP would keep track of when all incoming dependencies have arrived and when all messages have been sent out for a particular node. The RPP would also be responsible for sending update messages in the background. After each node is ready to be sent out, the processor would send a single message to the RPP, which would then send the dependent data to the correct processors. Additional background processing would be possible as the RPP could send update messages for nodes in the edge-loop while the processor is doing computation in the face-loop.

Overall the point of this section has been to emphasize that RPP benefits are not isolated to *appbt* alone. Rather, we feel that this approach shows significant promise as a low-cost accelerator in systems that provide no dedicated hardware cache coherence support.

8. Related Work and Discussion

Much of the closely-related work in the area of memory coherency protocols has been discussed throughout this paper. In the area of configurable computing, there has recently been some research interest in coupling configurable logic with the network interface. McHenry et al. [22] have proposed an FPGA-based front-end processor that filters information to an ATM firewall host to ensure network security. Guillaud et al. [23] have proposed a communication interface board for PCs which incorporates a transputer, an FPGA, and a VRAM to implement reconfigurable high level communication services for distributed real-time data and multimedia communication. None of these approaches, however, have considered configurable network interfaces with parallel computing applications in mind.

Much of this paper has discussed the performance and cost benefits of a reconfigurable protocol processor. Another advantage that systems like the RPP have is that it is relatively easy to update the coherency protocols as applications or protocol strategies change over time. All that is required is modification of the VHDL design to suit the new communication patterns. Hard-wired coherency protocols like Stanford's DASH [24] do not allow this.

9. Conclusion

This paper presents a new architecture for implementing application-specific cache coherency protocols. This study (i) identifies a new, easily-adopted use of configurable hardware in mainstreams systems, and (ii) provides benchmark evaluations characterizing both communication behavior and whole program performance. We feel that our RPP architecture hits a performance/cost sweet spot between software and custom hardware approaches. With little additional hardware expense and design time, an RPP-style architecture could be implemented on many of today's high-speed network cards. Our approach speeds up communication time in the *appbt* application by a factor of 11X. For completeness, we have also considered whole-program performance, rather than just the individual protocol handlers; our whole-program performance improvements of 12% are also significant. In exploring other applications, we have identified several that show the potential for even more sizable performance improvements.

References

[1] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* (Nov. 1989), vol. 7, no. 4, p. 321-359.

[2] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proc. 21st Annual Int. Symposium on Computer Architecture*, pp. 325-337, April 1994.

[3] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. *COMP/CON Spring 95*.

[4] Babak Falsafi, Alvin R. Lebeck, et al. Application-Specific Protocols for User-Level Shared Memory. *Supercomputing '94*, November 1994.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE-Micro*, Vol. 15, No. 1, pp. 29-36, Feburary 1995.

[6] Angelos Bilas. Improving the Performance of Shared Virtual Memory on System Area Networks. Technical Report #TR-586-98, Princeton Computer Science Department, August, 1998.

[7] Cheng Liao, Dongming Jiang, Margaret Martonosi, Douglas W. Clark, and Liviu Iftode. Monitoring Shared Virtual Memory on a Myrinet-based PC Cluster. *12th ACM International Conference on Supercomputing (ICS)*. July, 1998.

[8] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. University of Wisconsin-Madison, August 31, 1995.

[9] Mark Heinrich, Jeffrey Kuskin, et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. *Proc. 6th Int. Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA, October 1994.

[10] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherency Protocols. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[11] Jack E. Veenstra and Robert J. Fowler. MINT Tutorial and User Manual. Technical Report 452, Computer Science Department, The University of Rochester, June 1993 (Revised August 1994).

[12] "PCI Local Bus Specification," PCI Special Interest Group, Hillsboro, Oregon, April 1993 (Revision 2.0).

[13] "Techniques for Increasing PCI Performance", Intel Co.,Sep. 1997.

[14] Wenjia Fang, Edward Felten, and Margaret Martonosi. Contention and Queueing in an Experimental Multicomputer: Analytical and Simulation-based Results. Technical Report #TR-508-96, Princeton Computer Science Department, January, 1996.

[15] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. TR RNR-91-002 Revision 2, Ames Research Center, January 1991.

[16] FPGA Express Version 2.0, Synopsys Co.

[17] Workview Office Version 7.3, Viewlogic Co.

[18] XACTstep Foundation Series F1.3 Software, Xilinx Co.

[19] D. E. Culler, A. Dusseau, et al. Parallel Programming in Split-C. *Supercomputing 93*, pages 262-273, November 1993.

[20] Satish Chandra, James R. Larus, Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? *6th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, Oct.1994.

[21] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. *Proc. 5th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, July 1995.

[22] J.T. McHenry, P.W. Dowd, F.A. Pellegrino, T.M. Carrozzi, W.B. Cocks. An FPGA-based coprocessor for ATM firewalls. *Proc. 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.

[23] J.-F. Guillaud, G. Ouvradou, J.-D. Laisne, J. Le Drezen, B. Weber. A PC/ATM interface accelerator using reconfigurable technology. *Proceedings of the SPIE*, vol. 2608, pp. 134-45. 1995.

[24] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A.Gupta, J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems* (Jan. 1993) vol.4, no.1, p. 41-61.