Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
INTEGRATION, the VLSI journal 40 (2007) 235–252
INTEGRATION the VLSI journal
www.elsevier.com/locate/vlsi

Coprocessor design to support MPI primitives in configurable multiprocessors

Sotirios G. Ziavras[a,*], Alexandros V. Gerbessiotis[b], Rohan Bafna[a]

[a]Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA
[b]Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA

Received 1 March 2005; received in revised form 15 October 2005; accepted 24 October 2005

Abstract

The Message Passing Interface (MPI) is a widely used standard for interprocessor communications in parallel computers and PC clusters. Its functions are normally implemented in software due to their enormity and complexity, thus resulting in large communication latencies. Limited hardware support for MPI is sometimes available in expensive systems. Reconfigurable computing has recently reached rewarding levels that enable the embedding of programmable parallel systems of respectable size inside one or more Field-Programmable Gate Arrays (FPGAs). Nevertheless, specialized components must be built to support interprocessor communications in these FPGA-based designs, and the resulting code may be difficult to port to other reconfigurable platforms. In addition, performance comparison with conventional parallel computers and PC clusters is very cumbersome or impossible since the latter often employ MPI or similar communication libraries. The introduction of a hardware design to implement directly MPI primitives in configurable multiprocessor computing creates a framework for efficient parallel code development involving data exchanges independently of the underlying hardware implementation. This process also supports the portability of MPI-based code developed for more conventional platforms. This paper takes advantage of the effectiveness and efficiency of one-sided Remote Memory Access (RMA) communications, and presents the design and evaluation of a coprocessor that implements a set of MPI primitives for RMA. These primitives form a universal and orthogonal set that can be used to implement any other MPI function. To evaluate the coprocessor, a router of low latency was designed as well to enable the direct interconnection of several coprocessors in cluster-on-a-chip systems. Experimental results justify the implementation of the MPI primitives in hardware to support parallel programming in reconfigurable computing. Under continuous traffic, results for a Xilinx XC2V6000 FPGA show that the average transmission time per 32-bit word is about 1.35 clock cycles. Although other computing platforms, such as PC clusters, could benefit as well from our design methodology, our focus is exclusively reconfigurable multiprocessing that has recently received tremendous attention in academia and industry.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Configurable system; FPGA; Multiprocessor; MPI

1. Introduction

1.1. Overview of MPI

To facilitate parallel program development, the hardware can be abstracted as a collection of homogeneous processors, each with its own memory and support for interprocessor communications and synchronization. Such a distributed memory view is the default for PC clusters and the type of configurable architectures we envisage in Section 1.2. Parallel programming often follows the Single Program Multiple Data (SPMD) paradigm, where all processors run the same program on their own data sets. On top of the underlying hardware, software-based interprocessor communications can be realized via either shared memory or distributed memory concepts; the former leads to ease in programming whereas the latter to more optimized programming that focuses on program locality.

Interprocessor communications in distributed memory systems, such as PC clusters, come in two-sided and one-sided variants. The first variant employs an extensive

*Corresponding author. Tel.: +1 973 596 5651; fax: +1 973 596 5680.
E-mail address: ziavras@njit.edu (S.G. Ziavras).

0167-9260/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2005.10.001

collection of message-passing alternatives to realize two-way interprocessor communications. This variant first included the Parallel Virtual Machine (PVM) [1], and later various incarnations of Message Passing Interface (MPI) [2], such as LAM/MPI [3], MPICH, WMPI [4], that implement hundreds of functions, and some lesser known BSP (Bulk-Synchronous Parallel) libraries comprised of a few dozen functions, such as the Oxford BSP Toolset [5] and the Paderborn University PUB-Library [6]. Due to their size, these libraries are normally implemented in software, thus resulting in large communication latencies. A comprehensive study of the effects of these latencies on the performance of cluster architectures was presented by Martin et al. [7]. Significant improvements could be gained from a hardware implementation that can reduce communication latencies significantly, thereby increasing the overall bandwidth. However, the large number of functions in standard MPI makes any such implementation infeasible.

The one-sided Remote Memory Access (RMA) variant, on the other hand, is an effective way to program a distributed-memory parallel computer [8]. In this case, a single process puts data directly into another process's memory space. One-sided communication employs each time only a PUT or its symmetric GET instruction to communicate information. In contrast, message-passing implementation in MPI has more than eight choices. Thus, under RMA, the programmer does not have to choose among numerous alternatives for interprocessor communication as there is only one method to do so. Moreover, an optimized implementation of its communication library (in either software or hardware) is feasible due to its small size. RMA has actually been available since the introduction of the Cray SHMEM primitives [9] and more recently with the GASNet Extended Application Programming Interface (API) [10-12]. Despite the SHMEM implication in the name, RMA has been used widely for interprocessor communications in distributed memory systems. The simplicity of the RMA interface, and the elegance and efficiency of its primitives [13–17] led to its adoption, originally by some model-specific programming libraries, such as the Oxford BSP Toolset and PUB libraries, and subsequently by MPI-2 [8]. The current support of RMA in MPI libraries varies. Some freely available libraries, such as LAM/MPI and more recently MPICH, support it.

Table 1 shows a universal set of primitives required by the RMA framework we are proposing along with their MPI-2 equivalent functions [13]. Any other communications-related MPI function can be implemented using these primitives; operations such as gather, scatter, broadcast and scan can be implemented using PUT and potentially GET operations. BEGIN (*nprocs*), END ( ) and ABORT ( ) are the functions used to start, end and abort an SPMD program. NPROCS ( ) returns the number of processors in the present SPMD run. Each processor in the system has an identification number and PID ( ) returns that number locally. PUT ( ) and GET ( ) are the only functions used for data transfers. *PUT* (*dpid*, *srcaddr*, *desaddr*, *off*, *len*) is used by a given processor *spid* to send *len* words of data to processor *dpid*. The data are stored in location *srcaddr* in *spid*, while it is to be written starting at location *desaddr*+ *off* (*off* is an offset) in *dpid*. Symmetric to PUT ( ) is the GET ( ) operation. REGISTER (*desaddr*) and DEREGISTER (*desaddr*) provide a unique reference mechanism to address a remote (i.e., globally available) variable. In an SPMD program, the address of a variable that exists on multiple processors may not be the same across the distributed memories. If information is to be communicated from/into such a remote variable, a unique reference mechanism must be devised independently of the physical memory address of the variable; otherwise, each processor must maintain a collection of (local memory address, processor ID) pairs for the globally available variable. BARRIER ( ) is a synchronization operation that stalls the issuing processor until all processors in the system have also executed their BARRIER ( ) instruction.

Table 1
Primitives to support RMA

Primitive	Short description	MPI-2 equivalent function
BEGIN (*nprocs*)	Initiate an SPMD program on *nprocs* processors	MPI_Init
END ( )	Terminate the current SPMD run	MPI_Finalize
ABORT ( )	Abort	MPI_Abort
NPROCS ( )	How many processors?	MPI_Comm_Size
PID ( )	ID of issuing processor	MPI_Comm_rank
PUT (*dpid*, *srcaddr*, *desaddr*, *off*, *len*)	The processor, say with ID *spid*, sends the contents of its memory starting at address *srcaddr* of size *len* words to the processor with ID *dpid* to be stored into its memory starting at address *desaddr* + *off*.	MPI_Put
GET (*dpid*, *srcaddr*, *desaddr*, *off*, *len*)	Processor, say with ID *spid*, gets the contents of processor *dpid*'s memory starting at address *srcaddr*+*off* of size *len* words and copies them into its own memory starting at address *desaddr*	MPI_Get
REGISTER (*desaddr*)	Register *desaddr* as a globally available variable	MPI_Win_create
DEREGISTER (*desaddr*)	Deregister *desaddr*	MPI_Win_free
BARRIER( )	Stalls the calling processor until all other processors have executed their barriers as well	MPI_Barrier

1.2. Overview of configurable computing systems and motivation

Classical parallel computer systems based on multiple commercial-off-the-shelf (COTS) microprocessors cannot satisfy the high-performance requirements of numerous applications; not only are these requirements increasing at a rate faster than Moore's Law but the cost of these systems is prohibitively high. To bridge the performance gap but at a very high cost, Application-Specific Integrated Circuit (ASIC) implementations are often employed. However, ASIC designs are not flexible, require prohibitively high development costs and take very long time to market. FPGAs were once used in ASIC prototyping and glue logic realization for digital designs. In addition to the Xilinx and Altera companies, the most famous in FPGA production, virtually every chipmaker is pursuing reconfigurable computing nowadays, such as Hewlett-Packard, Intel, NEC, Texas Instruments, Philips Electronics and several startups. We use the general term configurable in our research to denote systems that are configured at static time and/or reconfigured at run time. Cray and SGI supercomputers engulfing FPGAs have recently become available as well. Each chassis in the Cray XD1 supercomputer contains six boards, each containing several processors and an FPGA [18].

Similarly, the experimental BEE2 system contains FPGAs for computations, and COTS components for memory and network interfaces [19]. Although an earlier BEE version targeted wireless communications simulation, BEE2 targets numerous high-performance applications. Finally, the FPGA High Performance Computing Alliance was launched in Edinburgh, UK, on May 25, 2005 [20]. Its purpose is to design and build a 64-node FPGA-based supercomputer capable of 1 TeraFlop performance; it will be located at the Edinburgh Parallel Computing Centre. The alliance members acknowledge the Herculean task of making this machine easy to program. The other members are Algotronix, Alpha Data, Institute for System Level Integration, Nallatech and Xilinx. To demonstrate its effectiveness, the alliance will port three supercomputer applications from science and industry. The alliance claims that this system will be up to 100 times more energy efficient than supercomputers of comparable performance and will occupy space equivalent to that of just four PCs. Of course, coprocessors of low-cost that could be embedded in FPGAs in the form of soft core IPs for implementing RMA primitives could benefit tremendously the latter two projects as well.

Parallel system implementations on a single FPGA were until recently exclusive for special-purpose, low-cost designs (e.g., systolic-array or data-parallel configurations) primarily due to resource constraints. However, it has been shown recently that FPGAs can now facilitate the implementation of programmable parallel systems inside one or more chips [21–24]. High versatility in programming is possible by often employing Intellectual Property (IP) soft processor cores, such as Nios from Altera Corp. and MicroBlaze from Xilinx Inc. Also, the implementation of floating-point units (FPUs) for processors embedded in FPGAs has been a recent trend due to the available higher resource density, as evidenced by Underwood [25] and Wang and Ziavras [24] and the introduction of MicroBlaze 4.00 in May 2005. The latter contains a 32-bit single-precision, IEEE-754 FPU which is part of the Xilinx EDK (Embedded Development Kit). FPGAs have relatively low cost so they can be used either to prototype or build the final product for parallel platforms. An FPGA-based evaluation board can cost from less than $500 to about $5000. The most advanced FPGA device costs a few hundred dollars, whereas a board with half a dozen to a dozen FPGAs may cost up to $100K; it is normally the software environment for the system that costs much more than the hardware. Due to the high cost of this software and the associated burdensome experience, any means of facilitating code portability across FPGA platforms in multiprocessor implementations (which are even more difficult to deal with) is absolutely essential.

To conclude, multi-million gate FPGAs make it now feasible to implement cluster-on-a-chip (multiprocessor) systems, thus providing reduced-cost platforms to utilize previously done research in parallel processing. Many dozens of specialized or a few dozen of semi-customized (i.e., programmable) computing processors may be embedded into a single FPGA using point-to-point networks such as those presented in [26,27]. Dally and Towles [26] introduced the concept of packet-transmitting on-chip networks to reduce the latency and increase the bandwidth in high-performance circuits. A methodology to design the interconnection network for multiprocessor systems on chips (MPSoCs) was presented by Benini and DeMicheli [27]. For example, we have embedded our own programmable processors in our HERA machine which was designed for matrix operations [23]; HERA implements partial reconfiguration at run time, where different groups of processors can execute simultaneously different pieces of Single Instruction Multiple Data (SIMD) or Multiple Instruction Multiple Data (MIMD) code, or a combination of these types of code. Despite HERA's wide versatility, each of the two target Xilinx XC2V6000 FPGAs contains about 20 processors. This number will increase significantly if this design is ported to more recently introduced FPGAs. Current FPGAs also have substantial support for I/O, which is crucial to parallelism. For example, not only does the Xilinx Virtex-4 FPGA also contain up to two PowerPC processors with interface for user coprocessors but it facilitates 1Gb/s Ethernet interconnectivity as well. Since intra-FPGA and on-board inter-FPGA direct communications are fast, communication overheads are drastically reduced when compared to PC clusters, the most popular parallel platforms. Additionally, these overheads could be reduced further by tailoring the configurable architecture to suit the communication patterns in the application.

Our main motivation is to support the portability of parallel code across FPGA-based systems embedded with

multiple soft microprocessor cores interconnected via on-chip direct networks. This has led us to the design of the communications coprocessor proposed in the current paper. This coprocessor implements the aforementioned MPI primitives (presented in Section 1.1) and, for the purpose of compatibility, communicates with any attached main processor via memory mapping techniques. Therefore, this coprocessor can support the data exchange requests of its attached processor as long as either RMA-based parallel programs are run or MPI function calls are converted by a compiler to use exclusively these primitives. Another advantage of this approach is that MPI-based code written for a PC cluster or a supercomputer could be potentially ported in a straightforward manner to such an FPGA-based parallel computing platform in an effort to provide portability across diverse architectures.

1.3. Related work

A few publications related to partial hardware support for MPI in cluster environments have appeared recently. However, such environments are not characterized by limited resources, so these solutions do not apply easily to FPGA-based designs. Despite this fact, a review of these projects follows and suitable performance comparisons with our design are presented in the experimental analysis section of this paper. An FPGA-based implementation of Sun MPI-2 APIs for the Sun Clint network is reported by Fugier et al. [28]. Clint was developed based on the observation that network traffic is often bimodal containing large and small packets requiring high throughput and low latency, respectively; it employs two physically separate channels for these two distinct types of packets. To improve the performance, new specifications for the implementation of MPI functions were provided. Gigabit Ethernet constitutes a low-cost solution in high-speed interconnects. On the other hand, advanced solutions, such as Scalable Coherent Interface (SCI), Myrinet, Quadrics and the Gigabyte System Network, improve the performance by integrating communication processors in the Network Interface Cards (NICs), realizing various user-level communication protocols and introducing programmability. Quadrics [29] goes one step further by integrating each node's local virtual memory into a globally shared virtual space; its Elan programmable network interface can deal with several communication protocols as well as fault detection and tolerance, whereas its Elite communication switches can be interconnected to form a fat tree. The complexity of Quadrics is such that it involves several layers of communications libraries. Quadrics is advertised to improve the performance of systems developed from high-complexity commodity server parts. In contrast, our target is configurable systems.

All complete implementations of MPI are software based. One such implementation over Infiniband is described by Liu et al. in [30], and a comparison of MPI implementations over Infiniband, Myricom and Quadrics is presented by Liu et al. [31]. A message-passing coprocessor is implemented by Hsu and Banerjee in [32] but it does not employ MPI. Our main objective is to enable the main CPU to offload all of its communication tasks to the coprocessor and not waste valuable CPU cycles in communication activities. A similar scheme is used in the IBM Blue Gene/L supercomputer, where the second processor in each node can act as a communications coprocessor [33]. The 1/2-roundtrip latencies on BlueGene are approximately 6 μs [34]. Myrinet was chosen by Tipparaju et al. [35] for an RMA-based study, despite its limited support for RMA (only the PUT operation is implemented in hardware); nevertheless, the results demonstrated performance improvement over MPI. Even this result justifies our effort.

Although modern network interfaces often offload work related to MPI processing, the time required to service requests stored in the receive queue may grow very substantially. Efforts to solve the problem through hashing do not provide good results because of increases in list insertion times and frequent wildcarded calls of two-sided MPI communication [36]. Based on this observation, a processing-in-memory (PIM) enhancement to an NIC that contains a PowerPC 440 embedded processor is proposed by Rodrigues et al. [36]. The PIM design decreases the search time for a linked list by carrying out multiple searches simultaneously. Realizing such a design with current silicon technology is not easy. Therefore, a theoretical analysis was presented of the required resources in the PIM design for a subset of 11 MPI functions. The design was simulated with a component-based discrete event simulator that integrated the SimpleScalar tool suite. The simulation involved a simple point-to-point, two-node network with fixed latency. An associative list matching structure to speedup the processing of moderate length queues under MPI is presented by Underwood et al. [37]. Simulations compared the performance of the resulting embedded processor to a baseline system. On the negative side, the pipeline in their FPGA-based prototype does not allow execution overlaps, therefore a new match is processed for every six or seven clock cycles. These two mentioned studies further intensify the importance of an RMA-based approach because of its reduced complexity and implementation efficiency [13].

This paper is organized as follows. The coprocessor design is presented in Section 2. It is implemented on a Virtex-II XC2V6000 FPGA. For the purpose of evaluating our approach, Section 3 shows the design of a fast router to which several coprocessors with their associated main CPUs can be attached to implement multiprocessor systems. Experimental results are covered in Section 4, while conclusions are presented in Section 5.

2. Coprocessor design

2.1. Registration Table (RT)

In a multiprocessor system, a globally available variable via which data can be communicated may reside in

different memory addresses of member processors. To make programming easy and efficient, this detail should be hidden from the user. Registration via MPI_Win_create in MPI-2 and Register( ) in our approach is a process that establishes a common name/referencing mechanism to a variable having multiple instances in an SPMD program. This local–global relationship is established through registration and can be handled by our coprocessor. Deregister( ) in our approach is a process opposite to registration, where information for a globally available variable is removed from all the local coprocessor tables. We use eight bits to represent *desaddr* in the registration process. Hence, information for up to $2^8 = 256$ globally shared variables may be present simultaneously. The local memory address is 32 bits wide, allowing the CPU to index 4 Gbytes of local memory; this space is more than enough for configurable multiprocessor designs. The registration operation stores the 32-bit local address of the globally available variable in the next vacant position of the local RT of every coprocessor in the multiprocessor; this position is the same in all the RTs. The 8-bit index to access the stored 32-bit address in the RT serves as the variable's global address and the deregistration operation removes the 32-bit local address from all the RTs in the multiprocessor.

The RT in our design is used to realize normal RAM lookup, Content Addressable Memory (CAM) lookup, and address register and deregister. In normal RAM lookup, given the 8-bit address of a globally available variable, the RT returns locally the 32-bit local address; this operation is applied when a packet is received from another coprocessor so that the requested local access can be accomplished with the correct address for the variable. In CAM lookup, given the 32-bit local address of a globally available variable, the RT returns the 8-bit index where the former address is stored; this operation is needed when the coprocessor executes a PUT or GET to a remote processor, and needs the 8-bit address of the former variable to form a packet. Thus, a combined CAM/RAM structure with very efficient read, write and delete operations is needed to implement the RT. The chosen building block for the RT is a 32-word deep, 9-bit wide CAM32 × 9 [38]. This design takes advantage of the dual-port nature of block SelectRAM + (BRAM) memories in Virtex-II devices. In addition to the distributed SelectRAM + memory that offers shallow RAM memories implemented in Xilinx Configurable Logic Blocks (CLBs), Virtex-II FPGAs also contain large BRAM blocks. There are 144 18 Kbit BRAM blocks in the XC2V6000 FPGA that we used to implement the coprocessor; they can store 2592 Kbits or 81K 32-bit words. The two ports A and B can be configured independently, anywhere from 16 K-word × 1-bit to 512-word × 32-bit. These ports have separate clock inputs and control signals, and access the same 16 Kbits of memory space with an addressing scheme based on the port width. Ports A and B are for write and read/match operations, respectively.

The unique feature of the CAM32 × 9 is that it decodes the 9-bit word in the input before storage (one input bit is connected permanently to 0 in our implementation since the index for RT access is 8 bits). As the input can assume the values 0–511, the input with value m can be represented with 511 zero bits and a single 1 bit in the mth binary position [38]. For example, for the binary input 000000101 (or 5 in decimal) the stored 512 bits are 000....000100000; the index of the least-significant bit is zero. M 512-bit words are then needed to store M decoded 9-bit words. Fig. 1 shows a 32 × 512 CAM array, for $M = 32$. When the 9-bit input is searched for in the CAM, the 32-bit port B displays the match(es). For our example in the figure, where the word 0000 00101 was previously written in column 2 of the CAM32 × 9, the match operation for the input with value 5 will return the 32-bit value 0000 0000 0000 0000 0000 0000 0000 0100 stored in row 5; it represents a match found in column 2 (the output is the decoded representation of 2). If no match is found, the output at port B is 0000 0000 0000 0000 0000 0000 0000 0000. For several matches, a 1 is returned for each corresponding column; of course, our RT design never needs to return more than one match.

For the write operation, the memory is viewed as a special type of RAM. The inputs to the CAM32 × 9 write port are the 9-bit data and a 5-bit address to access one of the 32 locations. Writing data into the corresponding location assumes first the transformation of the 9-bit input value into its 512-bit equivalent. However, the initialization to all zeros in the CAM32 × 9 macro allows to easily toggle just one bit in the 512-bit stored word for a change to 1. The 1-bit data and 14-bit address inputs are used by port A for writing; it is configured as 16384 × 1. For the write operation, the data input is asserted to 1, and the 9-bit data and 5-bit address inputs are merged to form the upper and lower fields in the 14-bit applied address. For the single-cycle erase of a CAM entry, we can use a write operation but with a 0 forced into storage instead of a 1. As earlier, the port A address input combines the 9-bit data to be erased and the 5-bit address. The CAM data are also stored in RAM32 × 1 primitives in an ERASE_RAM; these data are read back to erase the CAM. Nine RAM32 × 1 blocks are used to implement the ERASE_RAM. This RT design implements the read/match in one clock cycle, while two clock cycles are consumed by the write and erase operations.

Four CAM32 × 9 s can be cascaded to obtain the 32-bit wide CAM32 × 32 used for the RT; a match is possible only when all four CAMs match their data [38]. The RT table that was designed with the CAM32 × 32 block is shown in Fig. 2. The 32-bit STATUS register keeps a record of whether each position is filled or empty in the CAM32 × 32. A priority encoder generates the next address for a registration (i.e., write) operation by encoding the STATUS register. The output of the priority encoder is the index of the vacant position in the RT to be filled next time. The RT controller block is responsible for generating

ADDR[4:0]

	31	30	29	…	...	…	...	...	…	...	…	...	05	04	03	02	01	00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
3	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
4	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
5	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	**1**	0	0
.	…	...	…	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	…	...	…	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	…	...	…	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	…	...	…	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	…	...	…	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
.	0	0	0	…	...	…	...	...	…	...	…	...	0	0	0	0	0	0
510	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
511	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 1. CAM32 × 9 lookup example.

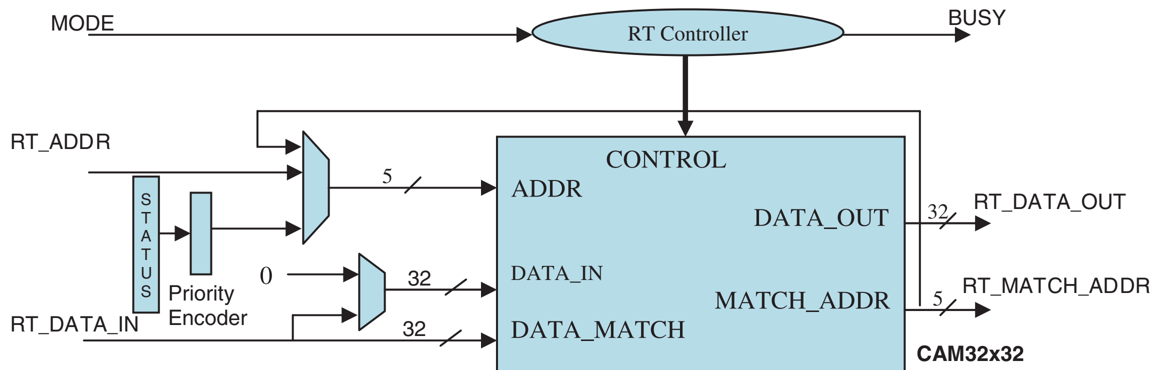


Fig. 2. Registration table.

the control signals for the two multiplexers and the CAM32 × 32, and also for asserting the BUSY signal during the multicycle register and deregister operations. Whereas registration is a write operation into the CAM, deregistration first requires a match operation to get the 5-bit globally available address, which is then used to perform an erase operation (by writing a 0) into the CAM. Table 2 summarizes all possible operations that involve the RT.

2.2. Coprocessor design

2.2.1. Coprocessor interface and architecture

The coprocessor architecture is shown in Fig. 3. The five stages in its pipeline are Instruction Fetch (IF), Instruction Decode (ID), Operand Read (OR), Execute1 (EX1) and Execute2 (EX2). The execution of PUT through the pipeline consists basically of forming the PUT packet header and then loading the transfer task information into the MM Interface FIFO. The execution of GET consists of forming the two-word GET packet header; the packet formats are presented in Section 2.2.3. The PUT and GET instructions consume two cycles in this process, whereas all the others consume a single cycle.

Using a Chip Select (CS) signal, which is produced by decoding the upper part of the CPU-issued address, the system designer can map the coprocessor to a desired address in the system memory map. The two least-significant bits A0 and A1 of the address bus along with the correct CS signal select a resource inside the

Table 2
Registration table operations

Mode	Operation	Input port	Output port	Clock cycles
00	NOP	—	—	1
01	RAM_lookup[a]	RT_ADDR[4:0]	RT_DATA_OUT[31:0]	1
	CAM_lookup[a]	RT_DATA_IN[31:0]	RT_MATCH_ADDR[4:0]	
10	Registration	RT_DATA_IN[31:0]	—	2
11	Deregistration	RT_DATA_IN[31:0]	—	3

[a]The RAM_lookup and CAM_lookup operations can be performed simultaneously in mode 01.

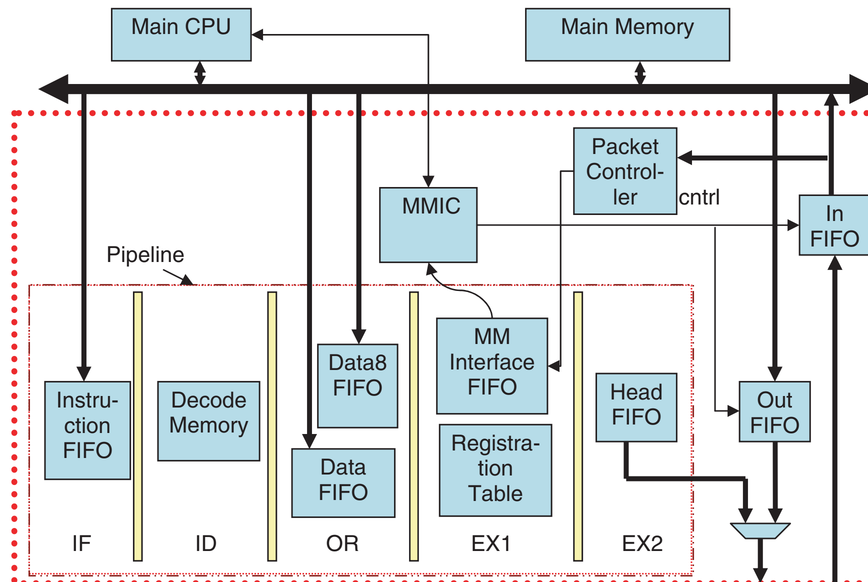
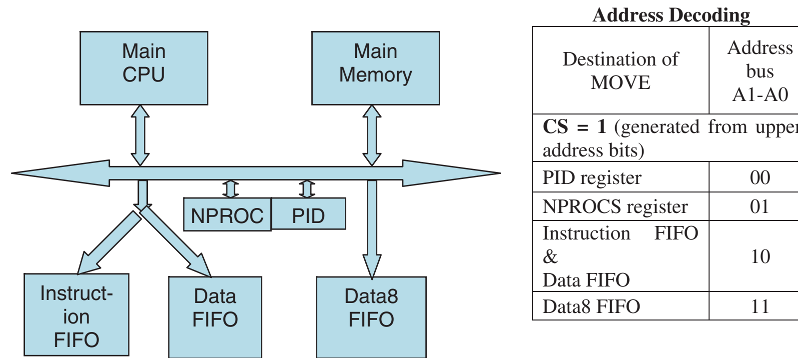


Fig. 3. Coprocessor architecture (MMIC: MM Interface Controller).

coprocessor. The processor can use simple MOVE commands to transfer data to three coprocessor FIFO buffers, namely the Instruction FIFO, Data FIFO and Data8 FIF0. An asynchronous FIFO is one where data are written into it from one clock domain and data are read from it using another clock domain. All the FIFOs in our design are asynchronous. This scheme makes the coprocessor portable and easy to interface to any CPU, a feature of paramount importance in configurable multiprocessor designs. In relation to CPU-coprocessor communication, it requires neither a change in the main CPU design nor in the compiler for programs utilizing the coprocessor. The compiler only has the trivial task of mapping each RMA instruction to one or more MOVE instructions. The coprocessor's direct access to the system's main memory enables it to do RMA transfers for which it is intended. The aforementioned FIFO structures serve as the instruction and data memories of the coprocessor and ensure that all the instructions can be smoothly pipelined in the coprocessor; this will become clear when the workings of the coprocessor pipeline are discussed. The decoding that selects the coprocessor FIFOs as well as the NPROC and PID registers is shown in Fig. 4. The latter two registers are also part of the coprocessor. The former register is written with the result of the NPROCS( ) primitive in Table 1. The latter register is written by the attached CPU with the node/processor ID. For the sake of simplicity, these two registers are ignored in the remaining figures.

With 32-bit address and data buses, to be able to transfer simultaneously to the coprocessor 32 bits of data and an opcode, part of the address bus should carry the opcode. The chosen encoding of coprocessor instructions is shown in Fig. 5. The two versions of PUT, namely PUT-1 and PUT-n, are for transferring one and n words, respectively, where $n>1$. This distinction makes possible a more efficient implementation of PUT-1. Based on this instruction encoding, Table 3 lists the operands sent to the coprocessor by its attached CPU for each RMA instruction. An individual MOVE instruction is needed to transfer each operand to the coprocessor. The opcodes, 32-bit operands and 8-bit operands are written into the Instruction FIFO, Data FIFO and Data8 FIFO, respectively (see Fig. 3).

All the RMA transfers to/from the main memory MM are handled by the Main Memory Interface Controller (MMIC) of Fig. 3. All the data to be written into the MM by the coprocessor are present in the In FIFO, while all the data that are copied from the MM into the coprocessor are



Address Decoding

Destination of MOVE	Address bus A1-A0
CS = 1 (generated from upper address bits)	
PID register	00
NPROCS register	01
Instruction FIFO & Data FIFO	10
Data8 FIFO	11

Fig. 4. Coprocessor interface and address decoding.

written into the Out FIFO. The fact that both the In FIFO and Out FIFO can be read and written asynchronously isolates this MM interface from the rest of the coprocessor pipeline. The MMIC is granted MM access by requesting it from the main CPU. Once it gets access, it executes each task stored as a triplet (direction of transfer, length of data, 32-bit starting address) in the MM interface FIFO. It continues executing tasks until there are no tasks pending or the main CPU requires its own MM access. This interface ensures that the coprocessor carries out MM transfers in the burst mode and does not try to get access to the MM on a per packet or instruction basis.

The packet controller in Fig. 3 handles the packets received from other coprocessors in the system. It either loads the MM Interface FIFO with an MM transfer task for a received PUT packet or causes the execution of a PUT into a remote processor for a received GET packet. The size of all the FIFOs in the coprocessor is user configurable. In our experiments, each of the Instruction FIFO, Data FIFO and Data8 FIFO is 64 words long; the In FIFO and Out FIFO are 512 words each, whereas the Head FIFO and MM Interface FIFO are 16 words each. Each word is 32 bits long. These choices represent a total of just 39 Kbits and do not affect the latencies inadvertently. The choice should, of course, be based each time on the overall parallel system size. The FPGA-based realization of the router in the VHDL language makes plausible such an adaptation.

2.2.2. *Barrier synchronization*

Whenever the main CPU encounters a barrier instruction, the coprocessor must complete all of its pending communication tasks and suspend operation until all the other coprocessors in the parallel system have done the same. In our design, the barrier instruction in the execution stage of the coprocessor pipeline activates the barrier mechanism. Due to the implicit in-order servicing of communication instructions by the coprocessor, all the instructions before the barrier synchronization have already gone through the pipeline. However, this does not imply that these instructions have been executed completely because of any of the following two reasons: requests related to these instructions may be waiting in the MM interface FIFO for data to be fetched from the main memory or may be queued in the Out FIFO for outgoing transmission. There may also be packets received from other coprocessors that are enqueued in the In FIFO waiting to be handled by the coprocessor's packet controller. Thus, for effective barrier synchronization, all coprocessor FIFOs must be empty and the packet controller must be idle. Naturally, the attached CPU does not load any instructions into the coprocessor after the barrier instruction until the coprocessor has signaled successful completion of the barrier. Once the coprocessor executes the barrier successfully, it asserts the Executed_barrier signal and waits for the HAVE_OTHERS_DONE signal to be asserted. The latter indicates that all the other coprocessors have reached their barrier as well and the system wide barrier is complete. In this barrier scheme, not only the coprocessors but the interconnection network participates as well. This is because of the simple reason that a packet might still be in the network while all the coprocessors have reached their barrier temporarily. So the barrier at the system level must take the network into consideration; a wait period of four clock cycles was implemented in our experiments because of the high efficiency and relatively small size of the networks in our experiments of Section 4. Each coprocessor now informs its attached CPU that the barrier has been reached by making the BARRIER_DONE signal true, which is acknowledged by the CPU by the BARRIER_DONE_ACK signal.

2.2.3. *Packet formats for communication*

The transmitted PUT-1 and PUT-*n* packets contain data, while the GET packet contains only information to enable the destination coprocessor to execute a PUT to the sending processor's main memory. The corresponding packet formats are shown in Fig. 6. The second word in the header of the GET packet is the header of the PUT packet that the receiving coprocessor has to create. The only change that the coprocessor has to make to produce the header of the PUT packet for a received GET packet is to increment the data length by one. The *dpid* field represents the destination processor's address, whereas

Register/Deregister

Data bus

Address
32 bits

Address bus

Opcode[1]	x	x	x
8	8	8	8

Set PID/NPROCS

Data bus

All 0`s	Value
24	8

Address bus

Opcode[1]	x	x	x
8	8	8	8

PUT – n / PUT – 1

Data bus

Destination address (*desaddr*)
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

Source address (*srcaddr*)
32

Opcode[1]	x	x	x
8	8	8	8

dpid	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Data length (*len*)	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Offset (*off*)	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Get

Data bus

Destination address
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

Source address
32

Opcode[1]	x	x	x
8	8	8	8

dpid	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Data length	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Offset	x	x	x
8	8	8	8

Opcode1[2]	x	x	x
8	8	8	8

Others

Data bus

All 0`s
32

Address bus

Opcode[1]	x	x	x
8	8	8	8

Fig. 5. Instruction encoding (x stands for "don't care" bits). [1]This field forces the opcode and the corresponding 32-bit data to be written into the Instruction FIFO and Data FIFO, respectively. [2]Opcode1 enables Data8 FIFO addressing where the 8-bit data is written temporarily into the Data8 Interface register. After every three writes into this register, a write is performed to the Data8 FIFO for the three 8-bit operands *dpid*, *len* and *off*.

the *myid* field in the GET packet is the address of the processor making the request. The 8-bit source and destination address fields are for globally available data, the packet length field represents the number of 32-bit words in the packet and the offset field follows the description of the PUT and GET primitives in Table 1. For compact packet encoding, the maximum packet length was chosen to be 31 words and, hence, the maximum data payload can be 30 words. Although a 2-bit field is sufficient to encode the packet type, a 3-bit field has been chosen instead to allow the possible addition of more packet types in the future. The encoding of the packet type is shown in Table 4.

3. Router design

To evaluate our coprocessor design, we have also implemented a very versatile network router that can be

used to interconnect coprocessors together for the formation of multiprocessor systems. Our router implementation basically consists of a packet switch having a crossbar architecture. The crossbar is the most rational choice because it can minimize the number of hops in packet transmission and, therefore, the actual performance of the coprocessor can be evaluated directly. In addition, the crossbar is a natural choice for on-chip configurable multiprocessor designs that may contain a few general-purpose or up to a few dozen specialized processors; they are the ultimate targets of our proposed approach. This choice is justified further by the reported consumed resources for its implementation, as described later in this section.

Table 3
Coprocessor operands

Instruction	Operands sent to the coprocessor
PUT-*n*, GET	Destination address, source address (both 32 bits); destination PID, length of data, offset (all 8 bits)
pUT-1	Destination address, actual data (both 32 bits); Destination PID, length of data, offset (all 8 bits)
Register/Deregister	Address (32 bits)
Set PID/NPROCS	Value (8 bits)
Begin, End, Abort, Barrier	None

There are three kinds of queuing in high-performance packet switches: input queuing, crosspoint queuing and output queuing. Golota and Ziavras [21] presented a crossbar router for parallel computers with unbuffered crosspoints, but buffered inputs and outputs. Virtual Output Queuing (VOQ) can be used to avoid blocking of the input queue because the receiving port for the head of the queue packet is full; separate queues are maintained in the input for different output ports. Rojas-Cessa et al. [39] proposed the Combined-Input-Crosspoint-Output Buffered (CIXOB-k) switch model where the crosspoint buffer has k-cell size, with VOQs at the inputs; simple round-robin is used for input and output arbitration. CIXOB-k with round-robin arbitration provides 100% throughput under uniform and unbalanced traffic.

We have implemented a crossbar router consisting of input VOQ ports as shown in Fig. 7. For an N-processor system, the crossbar consists of N^2 crosspoints arranged in N rows by N columns. Crosspoint CP (i, j) connects input i to output j. Each crosspoint in our implementation contains an asynchronous FIFO. The input VOQ port transmits the destination/output port number along with

PUT-1 packet format

dpid 8	Packet type 3	Packet length 5	Offset 8	Destination address 8	Data 32
Header (32-bit)					Data

PUT-n packet format

dpid 8	Packet type 3	Packet length 5	Offset 8	Destination address 8	Data1 32	………. 32	DataN 32
Header (32-bit)					Data (Number of 32-bit words: Packet length – 1)		

GET packet format

dpid 8	Packet type 3	Packet length 5	Offset 8	Source address 8	myid 8	Packet type 3	Data length 5	all 0`s 8	Destination address 8
Header1 (32-bit)					Header2 (32-bit)				

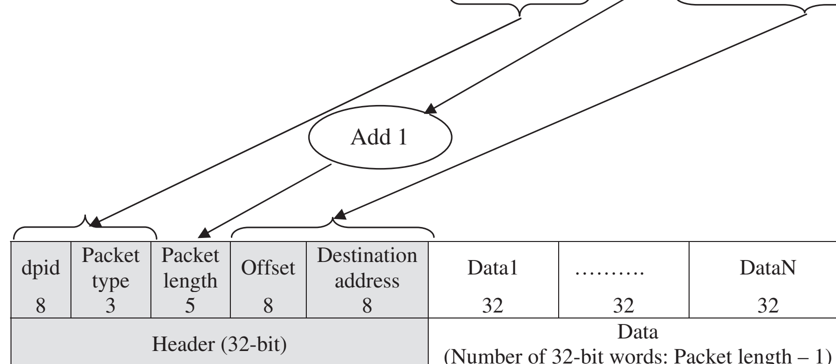


Fig. 6. PUT-1, PUT-*n* and GET packet formats.

the packet to the crossbar. This selects the crosspoint to which the packet is to be delivered. The status of the receiving crosspoint FIFO is sent back to the input VOQ port to aid the scheduler in that port in its task. A round-robin output scheduler selects one crosspoint from all the non-empty crosspoints on a column to transmit a packet to the corresponding output. The crosspoint FIFOs and the output memory FIFOs can be configured by the user; all are 32 words long in our implementation. In addition to the 32-bit data bus that connects a coprocessor to the router, two additional signals are included for handshaking.

Table 4
Packet type encoding

Packet	Encoding
PUT-1	001
PUT-*n*	010
GET	011

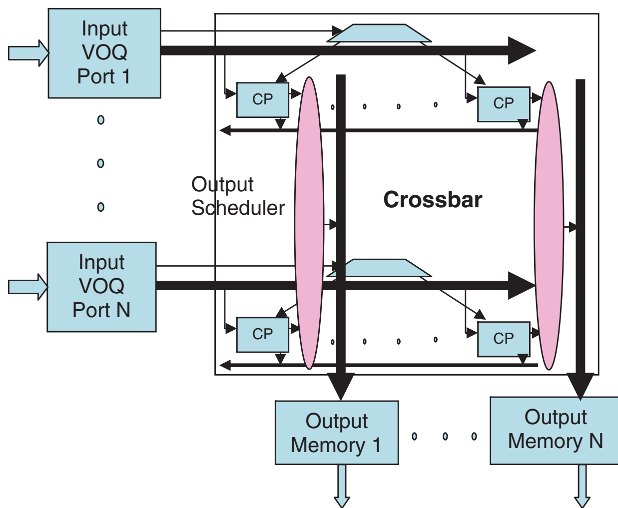


Fig. 7. Router architecture.

The basic organization of the input VOQ port is shown in Fig. 8. The incoming packets are queued in the Input Memory FIFO, and the part of the header consisting of the destination address and the length of the packet is queued in the Header Q FIFO. The VOQ controller employs an asynchronous request-grant mechanism to read the destination port for a packet from the routing table using the destination address stored in the Header Q FIFO. A packet destined for output i gets buffered in VOQ(i). Round-robin arbitration is used to select one VOQ from all the eligible VOQs to send a packet to the crosspoint buffer. An eligible VOQ is one which is not empty and for which the corresponding crosspoint buffer is not full. The status of the crosspoint buffers is determined by a feedback mechanism. The size of the Input Memory, Header Q and VOQ FIFOs is user configurable and equal to 512, 16 and 32 words, respectively, in our implementation.

In order to implement system-wide barriers as outlined earlier, a mechanism must be built to indicate the presence of packets in the router. The packets in the router can be either in one of the VOQ ports or in the crossbar. A packet in a VOQ port can be either in the INPUT MEMORY or in one of the N VOQ FIFOs. The ALL_FIFOs_EMPTY signal indicates whether all these FIFOs are empty. Similarly, the crossbar output ALL_FIFOs_EMPTY signal indicates that no packet is enqueued in the crossbar. Finally, a simple hardwired AND operation on these two signals generates the ROUTER_BARRIER signal which is used in the system-level implementation of barrier synchronization.

4. Implementation and experimental results

4.1. Target system for the experiments

The WILDSTAR II development board from Annapolis Micro Systems was used in our VHDL-based implementation of the coprocessor and router [40]. This board contains two Xilinx XC2V6000 speed grade 5 FPGAs with 128 MB

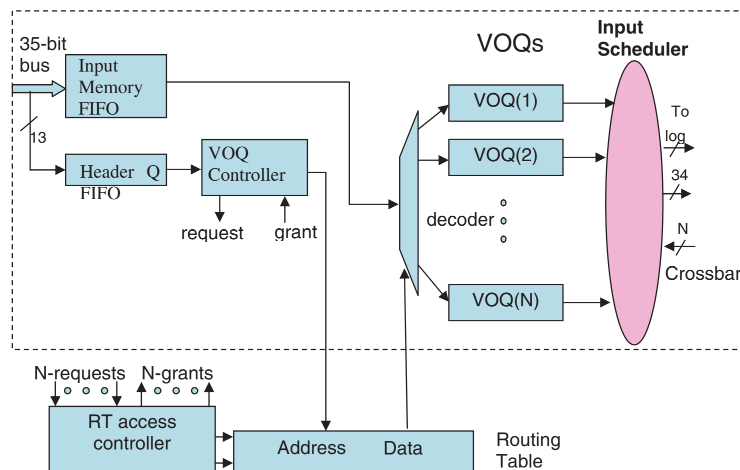


Fig. 8. Input VOQ port architecture.

of SDRAM distributed in two banks and accessed at 1.6 GBytes/s, and 24 MB of DDR II SRAM distributed in 12 banks and accessed at approximately 11 GBytes/s. The host PC communicates with the board via a 133 MHz PCI interface having a 64-bit wide data path. Each FPGA contains six million equivalent system gates. The XC2V6000 FPGA contains a 96×88 array of Configurable Logic Blocks (CLBs). Each CLB contains four slices, for a total of 33,792 slices. Each slice contains, among others, two 4-input function generators. Each generator is programmable as a 4-input Look Up Table (LUT), 16 bits of distributed SelectRAM+ (DI-RAM) or a 16-bit variable-tap shift register. Therefore, the maximum size of the DI-RAM can be 1.056 Mbits. Actually, single- or dual-port RAMs can be implemented using the DI-RAM. In the dual-port configuration, DI-RAM memory uses one port for synchronous writes and asynchronous reads, and the other port for asynchronous reads. The dual-port 16×1, 32×1 and 64×1 configurations consume 2, 4 and 8 LUTs, respectively. As mentioned in Section 2, the FPGA also contains 144 18 Kbit SelectRAM+ blocks (BRAMs).

A brief summary of our FPGA design flow follows. Coding and compilation were done using ModelSim from Mentor Graphics. The functional simulation was performed in Modelsim. The VHDL files were the input to the Synplify Pro synthesis tool. During synthesis, the behavioral description in the VHDL file was translated into a structural netlist and the design was optimized for the Xilinx XC2V6000 device. This generates a netlist in the Electronic Design Interchange Format (EDIF) and VHDL formats. The output VHDL file from the synthesis tool was used to verify the functionality by doing post synthesis simulation in Modelsim. The netlist EDIF file was given to the implementation tools of the Xilinx ISE 6.3i. This step consists of translation, mapping, placing and routing, and bit stream generation. The design implementation begins with the mapping or fitting of the logical design file to a specific device, and is complete when the physical design is completely routed and a bitstream is generated. Timing and static simulations were carried out to verify the functionality. This tool generates an X86 file which was used to program the FPGA. Then a C program was used; various standard API functions available by Annapolis Micro systems were used for communication between the host system and the board. During execution of this program, the host CPU programs the FPGA using the available X86 format file, writes the program data to the coprocessor(s) in the system, and after a given delay reads the results back.

4.2. FIFO implementation and resource consumption

Our coprocessor implementation consumes only about 2% of an FPGA's resources; indicatively, it consumes 1379 out of the 67,584 available LUTs and six out of the 144 available BRAM blocks (these numbers include the 284 LUTs and four BRAM blocks consumed by the CAM memory). The percentage of consumed resources should be much smaller than 2% for a more recent Xilinx Virtex-4 FPGA; for example, the XC4VLX200 contains 178,176 LUTs and 336 BRAM blocks. For a comparison, the design in [37] uses from 9025 to 18,045 LUTs. Even if we integrate the BRAM block space into the DI-RAM space, the maximum space consumed by our design is equivalent to 8291 LUTs.

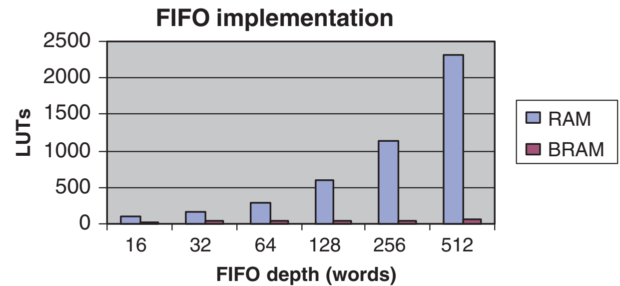


Fig. 9. Resource consumption by FIFO queues (RAM represents DI-RAM).

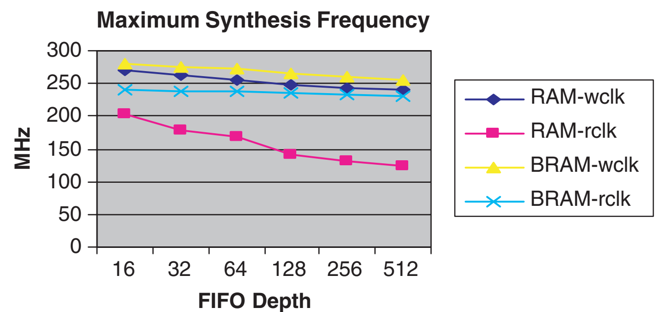


Fig. 10. Maximum synthesis frequency for FIFO queues (RAM represents DI-RAM; rclk: read clock; wclk: write clock).

The various FIFO queues in our design have been synthesized in one of two ways depending on their size; we use either DI-RAM or BRAM memory. Fig. 9 shows the space required for both styles of FIFO implementation with 32-bit words. In the case of BRAM, each FIFO queue also requires minor DI-RAM space. Fig. 10 shows the maximum frequency of the write (*wclk*) and read (*rclk*) clocks for each style of synthesis. DI-RAM FIFOs of more than 128 words deep use more than 1000 LUTs which is very high, and also their read clock frequency drops sharply below 150 MHz. Using BRAM for FIFOs of size less than 128 words wastes many resources since a single BRAM has 576 32-bit words capacity, and therefore, is partially used; there is only a small increase in the LUTs used that implement the read and write counters. Therefore, FIFOs of size less than 128 words are implemented with DI-RAM. In contrast, FIFOs of size equal to or greater than 128 words are implemented with BRAM. Tables 5 and 6 summarize the related choices.

A BRAM was used as the main memory with one port connected to the coprocessor via the coprocessor address

and data (CAD) bus, and the other port connected to the host CPU via the local address and data (LAD) bus. The main memory size is 512 words and is mapped to the CAD bus addresses 0 to 511. In our experiments, the bus interface logic consists of two FIFOs, the address bus FIFO and the data bus FIFO, each 512 words in size, as shown in the coprocessor test module (CTM) of the two-coprocessor system in Fig. 11. These two FIFOs are used to emulate the execution of MOVE commands that transfers opcodes to the coprocessor. The host program preloads the 32-bit MOVE destination address into the address bus FIFO and the corresponding data into the data bus FIFO. The control logic (not shown in the figure) reads both these FIFOs simultaneously via the CAD bus and generates the necessary write signal, thus emulating the execution of a MOVE. The control logic also serves as an arbiter for the CAD bus because this bus is used for two purposes—the execution of "virtual" MOVEs from the interface FIFOs to the coprocessor, and data transfer between the main memory and the coprocessor. A global system enable controls the interface logic. This global system ensures that all the coprocessors in a system start their program execution simultaneously. A 10-bit system counter (not shown in Fig. 11) is used for debugging and timing analysis purposes. The address bus FIFO, the data bus FIFO, the main memory and the system counter are memory mapped into the LAD bus memory.

Table 5
Coprocessor memory configuration

Module	Size (32-bit words)	Implementation type
Instruction FIFO	64	DI-RAM
Data FIFO	64	DI-RAM
Data8 FIFO	64	DI-RAM
Head FIFO	16	DI-RAM
MM Interface FIFO	16	DI-RAM
In FIFO	512	BRAM
Out FIFO	512	BRAM

Table 6
Router memory configuration

Module	Number	Size (32-bit words)	Implementation type
Input VOQ	N^2	16	DI-RAM
Crosspoint Buffer	N^2	16	DI-RAM
Input Memory	N	512	BRAM
Output Memory	N	16	DI-RAM

4.3. Two-coprocessor test system

Fig. 11 shows a two-coprocessor test system that we have implemented on the Annapolis board. Each CTM consists of the coprocessor, a BRAM as the node main memory and the interface logic that emulates the MOVE commands as described earlier. The packets being sent and received are monitored using Debug Memory 1 and 2. The system frequency is 50 MHz. The host C program loads the "application" program into the interface logic of each CTM and then triggers the System Enable signal that begins the execution of the "application" program on both coprocessors simultaneously. The 10-bit system counter starts counting at the same time and stops when the barrier is reached, thus giving the total time required for the run. Each instruction was run many times to calculate the average time required for the completion of an individual instruction. In each case, a single instruction was executed on coprocessor-1 followed by a barrier, while only a barrier was executed on coprocessor-2. Table 7 shows the measured execution time of instructions. Since the barrier instruction consumes eight clock cycles, to get a good approximation of the actual execution time for each of the

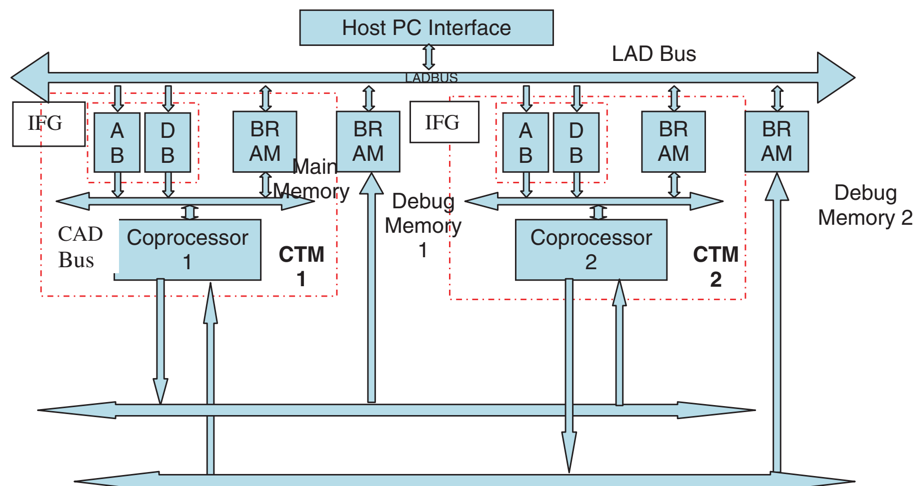


Fig. 11. A two-coprocessor test system (TCTS). AB: Address bus FIFO; DB: data bus FIFO; CTM: coprocessor test module; LAD: local address and data bus; CAD: coprocessor address and data bus. IFG: interface logic.

Table 7
Measured instruction execution times ($f = 50$ MHz)

Instruction	Clock cycles to barrier	Time to barrier (μs)
Barrier	8	0.16
Registration	10	0.20
Deregistration	11	0.22
PUT-8[a]	38	0.76
GET-8[a]	52	1.04

[a]GET-8 and PUT-8 refer to transfers of eight words of data.

Table 8
GET and PUT execution times

n	PUT-n	GET-n	PUT-n	GET-n
	Clock cycles *to barrier* (*measured*)	Clock cycles *to barrier* (*measured*)	Max. *calculated* clock cycles per instruction *without barrier*	Max. *calculated* clock cycles per instruction *without barrier*
1	31	44	18	31
2	32	46	19	33
4	34	48	21	35
8	38	52	25	39
16	46	60	33	47
30[a]	59	73	46	60

[a]The maximum data that can be sent in one packet is 30 words.

remaining instructions, eight clock cycles should be subtracted from the values listed in the table. In our experiments, transfer requests are injected into the coprocessor practically in every clock cycle as long as there is space in the data bus FIFO. A real CPU cannot make communication requests that often. Consider also the fact that application benchmarks as well make data transfer requests at a much lower rate. This implies that the coprocessors and interconnection network receive service requests in our experiments that correspond to much larger (than 2–4 node) systems.

To determine the execution time of GET and PUT for other message sizes, more work is needed. These instructions involve both processors. The first two columns in Table 8 show that GETs are much more expensive than PUTs due to their two-way nature. The last two columns in the table show the calculated execution time of individual instructions without barrier synchronization. To derive the calculated values corresponding to the involvement of the coprocessor fabric in the execution, we subtract from each measured entry one clock cycle consumed by the interface logic in Fig. 11, eight clock cycles needed for barrier synchronization inside a coprocessor, two clock cycles needed to combine the barrier results from the two coprocessors and the router (this is the minimum), and two clock cycles to transfer the data inside the crossbar (minimum as well). Therefore, the minimum of 13 clock cycles is subtracted from the entries in the first two columns of Table 8 in order to derive the execution time of individual instructions (for the involvement of the coprocessor fabric and without the barrier implementation).

Table 8 indicates that the one-word message transfer latency in our system is 31 and 44 clock cycles for a PUT-1 and a GET-1 instruction, respectively; it rises to 59 and 73 cycles for a PUT-30 and a GET-30 instruction, respectively. Although these figures include barrier synchronization as well, they represent much less than 1 μs delay for the first pair and no more than 1.50 μs for the second pair. This is also despite the fact that the chosen FPGA was introduced several years ago and does not have the higher speed of recent FPGAs. Our results compare favorably to the latency figures of 2.7–6.5 μs cited for Myrinet with GM drivers that bypass the operating system in the process of sending messages [41]. Pure latency figures in ([42], p. 12) for Quadrics/Elan and GASNet range from 3 to 8 μs. In ([42], p. 13) pure latency figures for Myrinet 2000 cards and GASNet range from 10 to 20 μs. Times of 20 and 33 μs are reported in ([42], p. 7) for the MPI-2 PUT and GET instructions on Myrinet, 8 μs for the PUT and GET instructions on QSNet Quadrics/Elan in [29], and 25 μs for PUT and GET with the Scalable Coherent Interface (SCI) in [43]. In addition, [43] cites 40–50 μs for Myrinet latencies of two-sided send/recv MPI communications, and also 40–50 μs latencies for the same two-sided primitives under Gigabit Ethernet. Therefore, our PUT and GET latency figures are much better than those reported in [42] for a variety of interconnects and one-sided instructions, and those reported in [44] for two-sided instructions. Very recent figures for the GASNet-based PUT and GET instructions [11] range from around 20 to 12–15 μs for MPI and GM on Myrinet, 20 and 10–14 μs for MPI and VAPI over Infiniband, 6–8 and 1–2 μs for MPI and Elan over Quadrics/QSNet2, 73–75 and 0.25–0.27 μs for MPI and SHMEM on a Cray X1, 12–13 and 0.02–0.04 μs for MPI and SHMEM on an SGI Altix, 82–83 and 38–43 μs for MPI and LAPI on an IBM Colony cluster, and 18–19 and 8–9 μs for MPI and LAPI on an IBM Federation cluster; LAPI is a low-level one-sided communication API library for IBM SP systems. Therefore, with perhaps the exception of a few dedicated hardware platforms, our figures are superior to the other interconnects. If we also consider that our implementation was done on an FPGA running at just 50 MHz, our design is of tremendous importance primarily in the configurable multiprocessors field. A 4 GHz ASIC implementation of our coprocessor could improve our reported latency figures 80-fold!

4.3.1. Total data exchange results

We further evaluated the coprocessor's performance using total data exchanges. In this test pattern, each processor sends H words of data to each other processor (including itself in our study). From the total time

Table 9
Measured total exchange performance (including barrier synchronization) and calculated effective bandwidth for the 2×2 system

H	Time for total exchange		g (μs/32-bits)
	Clock cycles	μs	
1	53	1.06	0.530
2	54	1.08	0.270
4	55	1.1	0.138
8	60	1.2	0.075
16	76	1.52	0.048
32	121	2.42	0.038
64	198	3.96	0.031
128	372	7.44	0.029
256	701	14.02	0.027
512	1396	27.92	0.027
1024	2771	55.42	0.027

Table 10
BSP-based analysis of the results

H	$\max\{L, gh\}$	Clock cycles (column 2 of Table 9)	$L+gh$
1	52	53	55
2	52	54	58
4	52	55	63
8	52	60	74
16	52	76	96
32	87	121	139
64	173	198	225
128	346	372	398
256	692	701	744
512	1383	1396	1435
1024	2765	2771	2817

measured, an estimate of the effective communication bandwidth is made. A total exchange was realized by a series of PUT-*n* instructions followed by barrier synchronization at the end. The total time required is denoted here by t. For the sake of simplicity, H is chosen to be a power of two and ranges in value from 1 to 1024 words. The parameter $g = t/(2^*H)$ is then calculated to find the average time required to transfer one 32-bit word in this hungry for resources communication pattern. The reciprocal of g approximates the effective bandwidth of the system. Increases in the value of H diminish the overall effect of the overheads corresponding to transmitting the header information and carrying out barrier synchronization at the end. We can deduce from Table 9 that for large values of H, the value of g converges to about 0.027 μs per 32-bit word transmission, which is very close to the ideal bandwidth since the system clock period is 0.02 μs (for 50 MHz). This shows that our coprocessor guarantees outstanding bandwidth.

4.3.2. Analysis of total exchange results

A BSP-based analysis is appropriate here as the results in Table 9 also include the barrier-synchronization time which is omnipresent in the model. The indicated parameter g is one of the BSP model [45]. Under the BSP model, a parallel platform is modeled as the triplet (p, L, g), where p is the number of processors, L is the synchronization periodicity of the system (the program is assumed to run in supersteps inside which processors compute without any communication and finally all synchronize via a barrier), and g is the cost of communication per word of data (i.e., the inverse of the router throughput). More formally, g is defined so that the cost of realizing a total exchange of size h (with each processor sending and receiving at most h words) in continuous message usage (i.e., for large h) is gh; for smaller values of h, the communication latency is bound and absorbed in parameter L that also includes the cost of barrier-style synchronization. Thus, one can assign the cost $max\{L, gh\}$ to a total exchange, where for the first term h corresponds to the maximum amount of information sent or received by any processor, and for the latter term $2h$ reflects the maximum amount of data sent or received. An alternative for the cost expression can be $L+gh$.

We will use both expressions in our BSP-based analysis, that is $max\{L, gh\}$ and $L+gh$ (an upper bound), to model our coprocessor. We claim that an analysis of Table 9 results under the BSP model implies $L = 52$ clock cycles (after subtracting one cycle from the first entry in Table 9 representing injection of the transfer request by the emulator) and $g = 1.35$ cycles per word (since g converges to about 0.027 μs and the clock cycle is 0.02 μs). Considering the H values in Table 9, the suggested L and g values, and noting that $h = 2H$, we obtain Table 10. The results in this table demonstrate that either of the two proposed BSP costs is quite close to the measured time, and in all cases the observed time is between the two BSP costs. Thus, this analysis proves that $g = 1.35$ clock cycles which is a very good value.

Benchmark programs for the total exchange operation based on the MPI primitives of Table 1 were developed by one of the coauthors [13]; some experimental results involving other benchmark programs were reported as well. The total exchange benchmark programs were run on a nine-node dual-processor cluster consisting of 1.2 GHz AMD Athlon CPUs, each one having 64K L1 and 256K L2 caches; the nodes were connected by a 1 Gbit/s Ethernet switch [13]. Similarly to our presented experiments, the results on the PC cluster involved two nodes, utilizing one CPU in each node. Fig. 12 compares the g parameter of the PC cluster and the two-coprocessor system. The results prove that our coprocessor implements the MPI primitives much more efficiently. The 1 Gbit/s Ethernet switch in the cluster and the crossbar in our 2×2 system do not have any significant negative effect on the performance, so the results in the figure show real node performance in implementing the MPI primitives.

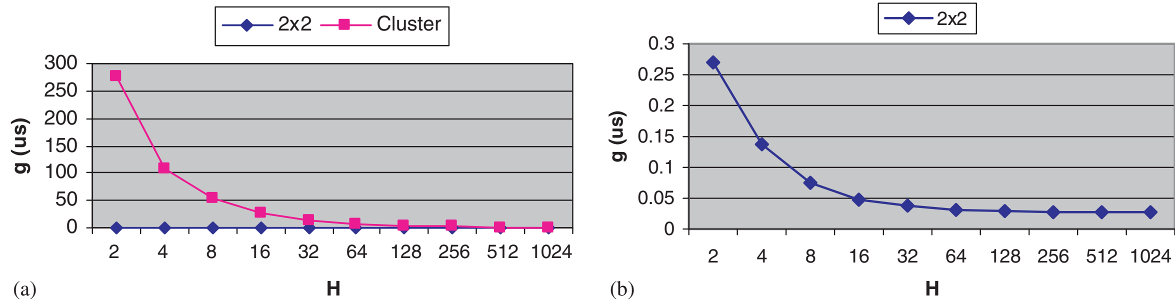


Fig. 12. (a) Comparison of g for the two-coprocessor (2×2) system and the PC cluster; (b) g for the 2×2 system plotted separately for higher precision.

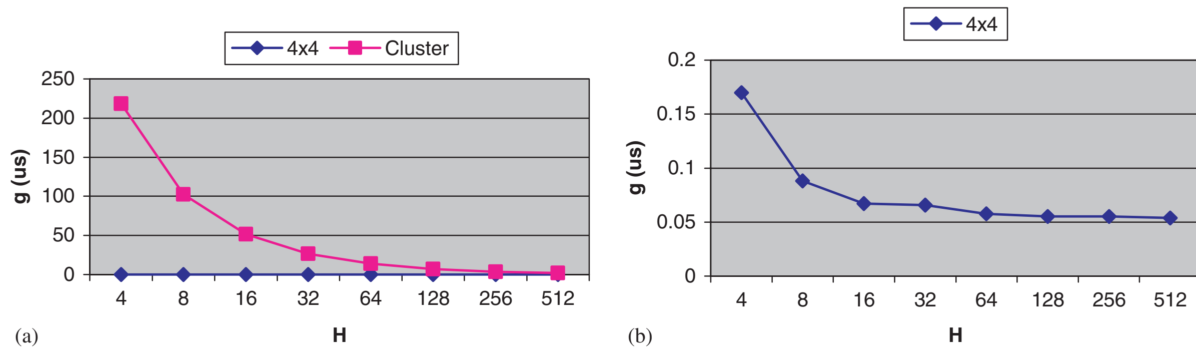


Fig. 13. (a) Comparison of g for the four-coprocessor (4×4) system and the PC cluster; (b) g for the 4×4 system plotted separately for higher precision.

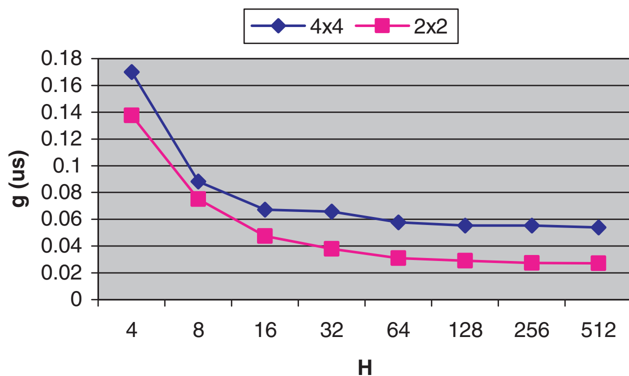


Fig. 14. Comparison of g for the 2×2 and 4×4 systems.

4.4. Four-coprocessor test system

Four-coprocessor test modules (CTMs) were interconnected as well using a 4×4 crossbar switch for total data exchange operations. A comparison in Fig. 13 with similar runs on the aforementioned PC cluster (results published in [13]) demonstrates the outstanding performance of the coprocessor. Fig. 14 shows a comparison of total exchange times for the 2×2 and 4×4 systems. The time for 4×4 is slightly larger than that for 2×2 primarily due to increased communication latencies in the larger crossbar and the larger probability of having simultaneously multiple messages in queues. The results prove once more the superiority of our design and its good scalability. The crossbar is often a good choice for on-chip systems containing up to a few dozen processors. To build even larger systems, such chips may be interconnected via a second-level network of the same or another type.

5. Conclusions

This paper demonstrated the design and implementation of a coprocessor for communicating messages in multiprocessor systems under the RMA model. A universal and orthogonal set of RMA primitives were implemented directly in hardware targeting a low-cost hardware design of low latency. Our research applies primarily to multiprocessor on-chip designs embedded in FPGAs. However, comparisons with other advanced approaches targeting more conventional computer platforms were presented as well. We also implemented a crossbar router that was used to build parallel systems for an experimental analysis of performance. The latencies for the studied one-sided communications are reduced dramatically as compared to those in a typical cluster environment. A 32-bit word transfer requires about 1.35 clock cycles on a Xilinx XC2V6000 FPGA running at 50 MHz. Introducing these RMA primitives in configurable computing creates a framework for efficient code development involving data exchanges independently of the underlying hardware implementation. A direct comparison with conventional parallel platforms without program rewriting is then also possible.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CNS-0435250 and IIS-0324816, and the Department of Energy under grant DE-FG02-03CH11171.

References

[1] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jaing, V. Sunderam, PVM: A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Boston, 1994.
[2] Message Passing Interface Forum, 〈http://www.mpi-forum.org/〉.
[3] LAM/MPI Parallel Computing, 〈http://www.lam-mpi.org〉.
[4] WMPI, Critical Software Inc., 〈http://www.criticalsoftware.com〉.
[5] J.M.D. Hill, W. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R. Bisseling, BSPlib: the BSP programming library, Parallel Comput. 2 (14) (1998) 1947–1980.
[6] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, The Paderborn University BSP (PUB)Library, Parallel Comput. 29 (2003) 187–207.
[7] R. Martin, A. Vahdat, D. Culler, T. Anderson, Effects of communication latency, overhead, and bandwidth in a clustered architecture, Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997, pp. 85–97.
[8] W. Gropp, E. Lusk, Advanced topics in MPI programming, in: W. Gropp, E. Lusk, T. Sterling (Eds.), Beowulf Cluster Computing with Linux. 2nd ed., MIT Press, Boston, 2003, pp. 245–278.
[9] R. Bariuso, A. Knies, SHMEM's User's Guide, SN-2516, Cray Research, Inc., Eagan, Minnesota, 1994.
[10] GasNet, 〈http://www.cs.berkeley.edu/~bonachea/gasnet/gasnet-sc03.pdf〉.
[11] C. Bell, D. Bonachea, W. Chen, J. Duell, P. Hargrove, P. Husbands, C. Iancu, W. Tu, M. Welcome, K. Yelick, An alternative high-performance communication interface, 18th Annual ACM International Conference on Supercomputing, Saint Malo, France, June–July, 2004. 〈http://www.cs.berkeley.edu/~bonachea/gasnet/gasnet-sc04-web.pdf〉.
[12] C. Bell, W. Chen, D. Bonachea, K. Yelick, Evaluating support for global address space languages on the Cray X1, 18th Annual ACM International Conference on Supercomputing, Saint Malo, France, June–July, 2004.
[13] A.V. Gerbessiotis, S.Y. Lee, Remote memory access: a case for portable, efficient and library independent parallel programming, Scient. Programm. 12 (3) (2004) 169–183 〈http://www.cs.njit.edu/~alexg/pubs/papers.html〉.
[14] R. Dohmen, Experiences with switching from SHMEM to MPI as communication library, Sixth European SGI/Cray MPP Workshop, Manchester, UK, 7–8 September 2000.
[15] G. Luecke, W. Hu, Evaluating the performance of MPI-2 one-sided routines on a Cray SV1, Technical Report, Iowa State University, December 21, 2002.
[16] G.R. Luecke, S. Spanoyannis, M. Kraeva, The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600, Concurrency Comput.: Practice Exp. 16(10) (2004) 1037–1060.
[17] D. Bonachea, GASNet specification, v1.1, University of California, Berkeley, Technical Report CSD-02-1207, October 29, 2002.
[18] Cray XD1 Supercomputer, Cray Inc., 〈http://www.cray.com〉.
[19] C. Chang, J. Wawrzynek, R.W. Brodersen, BEE2: a high-end reconfigurable computing system, IEEE Des. Test Comput. (2005) 114–125.
[20] Alliance Launched in Scotland to Develop the Next Generation of Supercomputers, FPGA High Performance Computing Alliance, May 25, 2005, 〈http://www.scottish-enterprise.com/sedotcom_home/stn/scottishtechnology/scottishtechnologynews/scottishtechnologynews-fhpca.htm〉.
[21] T. Golota, S.G. Ziavras, A universal, dynamically adaptable and programmable network router for parallel computers, VLSI Des. 12 (1) (2001) 25–52.
[22] X. Wang, S.G. Ziavras, Performance optimization of an FPGA-based configurable multiprocessor for matrix operations, IEEE International Conference on Field-Programmable Technology, Tokyo, Japan, December 15–17, 2003.
[23] X. Wang, S.G. Ziavras, Exploiting mixed-mode parallelism for matrix operations on the HERA architecture through reconfiguration, IEE Proceedings, Computers and Digital Techniques, accepted for publication.
[24] X. Wang, S.G. Ziavras, Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines, Concurrency Comput.: Pract. Exper. 16 (4) (2004) 319–343.
[25] K.D. Underwood, FPGAs vs. CPUs: trends in peak floating-point performance, 12th ACM International Symposium on FPGAs, Monterey, CA, February 22–24, 2004, pp. 171–180.
[26] W.J. Dally, B. Towles, Route Packets, Not Wires: On-chip interconnection networks, Design Autom. Conf., Las Vegas, NV, June 18–22, 2001, 684–689.
[27] L. Benini, G. DeMicheli, Networks on chips: a new paradigm for component-based MPSoC design, IEEE Computer (2002) 70–78.
[28] N. Fugier, M. Herbert, E. Lemoine, B. Tourancheau, MPI for the Clint Gb/s Interconnect, Proceedings of the 10th European PVM/MPI User's Group Meeting, October 2003, pp. 395–403.
[29] Quadrics QSNet Interconnect, Quadrics Ltd., 〈http://www.quadrics.com〉.
[30] J. Liu, J. Wu, S.P. Kini, P. Wyckoff, D.K. Panda, High performance RDMA based MPI implementation over InfiniBand, 17th Annual ACM International Conference on Supercomputing, June 2003.
[31] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, D.K. Panda, Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics, Proceedings of the Supercomputing Conference, November 2003.
[32] J. Hsu, P. Banerjee, A message passing coprocessor for distributed memory multicomputers, Proceedings of the Supercomputing Conference, November 1990, pp. 720–729.
[33] N.R. Adiga, et al., An overview of the Blue Gene/L supercomputer, Proceedings of the Supercomputing Conference on High Performance Networking and Computing, November 2002.
[34] G. Almasi, C. Archer, J.G. Castanos, C.C. Erway, P. Heidelberger, X. Martorell, J.E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-Burrow, W. Gropp, B. Toonen, Implementing MPI on the BlueGene/L Supercomputer, Europar, 2004.
[35] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, D.K. Panda, Exploiting nonblocking remote memory access communication in scientific benchmarks on clusters, International Conference on High Performance Computing, Bangalore, India, 2003.
[36] A. Rodrigues, R, Murphy, R. Brightwell, K.D. Underwood, Enhancing NIC performance for MPI using processing-in-memory, Workshop on Communication Architecture for Clusters, Denver, CO, April 4–8, 2005.
[37] K.D. Underwood, K.S. Hemmert, A. Rodriques, R. Murphy, R. Brightwell, A hardware acceleration unit for MPI queue processing, 19th International Parallel and Distributed Processing Symposium, Denver, CO, April 4–8, 2005.
[38] J.L. Brelet, L. Gopalakrishnan, Using Virtex-II Block RAM for High Performance Read/Write CAMs, Xilinx Application Note XAPP260 (v1.1), February 27, 2002, 〈http://xilinx.com/bvdocs/appnotes/xapp260.pdf〉.
[39] R. Rojas-Cessa, E. Oki, H.J. Chao, CIXOB-k: combined input-cross point-output buffered packet switch, Proceedings of IEEE GLOBECOM, November 2001, pp. 2654–2660.
[40] Wildstar II Hardware Reference Manual (revision 5.0), Annapolis Micro Systems. 〈http://www.annapmicro.com〉.
[41] Myrinet Overview, Myricom Inc., 〈http://www.myricom.com/myrinet/overview/〉.

[42] Distributed Shared-Memory Parallel Computing with UPC on SAN-based clusters, 〈http://www.hcs.ufl.edu/upc/HPNQ3ReportPPT95.ppt〉.

[43] H. Su, B. Gordon, S. Oral, A. George, SCI networking for shared-memory computing in UPC: blueprints of the GASNet SCI conduit, IEEE Workshop High-Speed Local Networks, Tampa, Florida, November 2004.

[44] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, Open MPI's TEG point-to-point communications methodology: comparison to existing implementations, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004. 〈http://www.open-mpi.org/papers/euro-pvmmpi-2004-p2p-perf/euro-pvmmpi-2004-p2p-perf.pdf〉.

[45] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.

Sotirios G. Ziavras received the Diploma in Electrical Engineering from the National Technical University of Athens, Greece, in 1984, the M.Sc. in Computer Engineering from Ohio University in 1985 and the Ph.D. degree in Computer Science from George Washington University in 1990. He was a Distinguished Graduate Teaching Assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He joined in Fall 1990 the ECE Department at New Jersey Institute of Technology as an Assistant Professor. He was promoted to Associate Professor and then to Professor in 1995 and 2001, respectively. He is an Associate Editor of the Pattern Recognition journal. He has published more than 100 research papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World and Who's Who in the East. His main research interests are advanced computer architecture, reconfigurable computing, embedded computing systems, parallel and distributed computer architecture and algorithms, and network router design.

Alexandros V. Gerbessiotis holds a Diploma in Electrical Engineering from the National Technical University of Athens (Greece), and S.M. and Ph.D. degrees from Harvard University. He is currently an Associate Professor with the Computer Science Department at New Jersey Institute of Technology. His research interests include parallel computing, architecture independent parallel algorithm design, analysis and implementation, sequential algorithms, experimental algorithmics, and graph theory.

Rohan Bafna received the M.Sc. in Computer Engineering from New Jersey Institute of Technology in June 2005 and the Bachelor of Engineering in Electrical Engineering from the Government College of Engineering, Pune, India, in 2001.