Computer Science Department
University of Crete

Software implementation of MPI primitives in multicore FPGA

Master's Thesis

Maria Katsamani

March 2010

Heraklion, Greece

University of Crete
Computer Science Department

Software implementation of MPI primitives in multicore FPGA

Thesis submitted by
Maria Katsamani
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

Author: ______________________________
Maria Katsamani

THESIS APPROVAL

Committee approvals: ______________________________

Manolis Katevenis
Professor, Thesis Supervisor

____

Dimitrios Nikolopoulos
Associate Professor, University of Crete

____

Angelos Bilas
Associate Professor, University of Crete

Departmental approval: ______________________________

Panagiotis Trahanias
Professor, Director of Graduate Studies

Heraklion, March 2010

Computer Science Department
University of Crete

Software implementation of MPI primitives in multicore FPGA

Master's Thesis

Maria Katsamani

Abstract

Chip Multiprocessors (CMP) are the dominant architectural approach since the middle of this decade. They integrate multiple processors on a single chip. However, it is still not obvious how to develop software that exploits the amounts of hardware and the operations available in such platforms to satisfy requirements for functionality and high performance communication.

Messaging layer software that adheres to Message Passing Interface (MPI) standard specifications has been very popular for almost two decades. The MPI standard fundamentally provides an abstraction layer that captures common application communication requirements. Whenever an abstraction layer is used, performance might be less than optimal. However, abstractions typically assist programmers to determine software requirements, while also providing a portability path for existing applications designed and implemented with an abstraction layer in mind. Abstraction layers are a natural ingredient of the software development cycle exploited for fast deployment of new features. The use of abstraction layers might be minimized at later stages of development, if performance issues associated with them are determined.

This thesis designed and implemented commonly used MPI primitives over a an FPGA prototyping platform that includes multiple MicroBlaze processors. We implemented features introduced in previous theoretical and practical work in the field of MPI standard implementations and continued the establishment of a software code base targeted to the particular FPGA prototyping platform. The implementation demonstrates how the underlying hardware operations can be exploited to develop a software messaging layer and facilitate high performance communication support among processors in a CMP environment.

Supervisor professor: Manolis Katevenis

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Υλοποίηση με λογισμικό βασικών λειτουργιών MPI σε πολυπύρηνη FPGA

Μεταπτυχιακή Εργασία

Μαρία Κατσαμάνη

Περίληψη

Τα (**chips**) Πολυεπεξεργαστικών Συστημάτων αποτελούν τη βασική αρχιτεκτονική τάση από τα μέσα της δεκαετίας. Ενσωματώνουν πολλαπλούς πυρήνες επεξεργασίας στο ίδιο **chip**. Όμως, δεν είναι ακόμα προφανές με ποιο τρόπο θα πρέπει να αναπτύσσεται λογισμικό που να εκμεταλλεύεται το πλήθος του υλικού και τις λειτουργίες που παρέχονται από το υλικό σε αυτά τα συστήματα, προκειμένου να ικανοποιηθούν απαιτήσεις λειτουργικότητας και επικοινωνίας με υψηλές επιδόσεις.

Το λογισμικό επικοινωνίας που συμμορφώνεται με τις προδιαγραφές του προτύπου **Message Passing Interface (MPI)** παραμένει δημοφιλές για σχεδόν δύο δεκαετίες. Το πρότυπο **MPI** παρέχει ένα επίπεδο αφαίρεσης, το οποίο περιλαμβάνει συνηθισμένες ανάγκες επικοινωνίας μεταξύ εφαρμογών. Κάθε φορά που χρησιμοποιείται ένα επίπεδο αφαίρεσης, η επίδοση μπορεί να είναι μικρότερη από τη βέλτιστη. Όμως, τα επίπεδα αφαίρεσης τυπικά βοηθούν τους προγραμματιστές να κατανοήσουν τις απαιτήσεις από το λογισμικό, ενώ παρέχουν και μια διαδικασία να εξασφαλίζεται η διαλειτουργικότητα για τις εφαρμογές που έχουν ήδη αναπτυχθεί ώστε να χρησιμοποιούν ένα επίπεδο αφαίρεσης.

Σε αυτή την εργασία σχεδιάστηκαν και υλοποιήθηκαν συχνά χρησιμοποιούμενες βασικές λειτουργίες του **MPI** σε ένα πρωτότυπο βασισμένο σε **FPGA** που περιέχει πολλαπλούς πυρήνες επεξεργασίας **MicroBlaze**. Υλοποιήσαμε χαρακτηριστικές λειτουργίες που έχουν προταθεί σε προηγούμενη θεωρητική και πρακτική μελέτη στο πεδίο των υλοποιήσεων του **MPI** και συνεχίσαμε την ανάπτυξη λογισμικού για το συγκεκριμένο πρωτότυπο σε **FPGA**. Η υλοποίηση επιδεικνύει τον τρόπο που μπορούν να χρησιμοποιηθούν οι λειτουργίες του υλικού για την ανάπτυξη λογισμικού επικοινωνίας το οποίο να διευκολύνει την επικοινωνία με υψηλές επιδόσεις μεταξύ επεξεργαστών σε ένα περιβάλλον πολυεπεξεργαστικών συστημάτων.

Επόπτης καθηγητής: Μανόλης Κατεβαίνης

Acknowledgements

I am grateful to my supervisor, Prof. Manolis Katevenis for offering this work to me, for his valuable assistance and for his support during the low times.

I would like to thank Dimitrios Nikolopoulos for all his guidance through this work. I would also like to thank Manolis Marazakis for his help during an earlier stage of this work in another hardware platform. Moreover, I would like to thank Angelos Bilas for his valuable insights while I was attending the course Parallel Computer Architecture.

Special thanks to Vassilis Papaefstathiou, Stamatis Kavadias and Giorgos Nikiforos for all their help with theoretical and practical issues, as well as to all the fellow members of the CARV Laboratory of FORTH-ICS for our discussions and their encouragement.

Finally, I would like to thank my family and friends for their continuous support.

This work was carried out with technical support from FORTH-ICS.

Maria Katsamani
Heraklion, March 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers automate procedures that occur in everyday life of humans. Since their inception, there have been ongoing efforts to produce faster systems, in order to extend further the benefits of computer usage in all areas of economic and social development. Sometimes performance improvements lead to the introduction of new computer applications that were not possible before. Other times the demands of existing or desired applications lead to increased efforts for performance improvements. In any case, there exists a feedback cycle between computer industry and the wider audience.

Although there have been performance improvements due to novel architecture ideas, such as pipelining and the introduction of caches [2], the performance of computer systems for many decades has improved mainly due to advances in the transistor technology used to built the hardware of such systems.

Limitations in further speedup of the clock frequency of processors became apparent during the last decade. The difficulty in overcoming those limitations intensified efforts to improve performance with rearrangements of the hardware architecture. Once again, improvements in hardware technology made viable, more than ever before, the introduction of multiple processors on a single chip.

Parallel software was being developed for decades, especially in the context of high-performance clusters. An increased current need for parallel software development is a natural consequence of the fact that alleged high-performance hardware is being shipped with multiple processors on a single chip. The hardware on its own cannot deliver the expected performance unless proper software that exploits its increased resources is also deployed. Typical questions that arise include how to quickly develop software that exploits new hardware organization and operations from a functionality perspective, how to port existing applications to the new architectures, and how to fine-tune the software to deliver performance close to the hardware capabilities.

As multiple processors are being deployed and available work is distributed among them, communication between the processors is typically needed for coordination and feedback purposes. The more often communication is needed, the more important communication performance becomes. Part of the steps required for communication are typically assigned to software components, while other steps are automatically performed by the hardware after communication initiation by the software. The software components that perform tasks related to communication define the messaging layer software. Such software typically prepares data for transmission (e.g. prepare packet headers) and control network interfaces using a predefined protocol that the hardware understands and expects. An overview of messaging layer software, its typical functionality and descriptions of messaging layer implementations such as Active Messages and Illinois Fast Messages can be found in [1]. [1] also includes an overview of Message Passing Interface (MPI).

Messaging layer software that adheres to the MPI standard specifications has been very popular for more than two decades. The MPI standard fundamentally provides an abstraction layer that captures common application communication requirements. Whenever an abstraction layer is used, performance might be less than optimal. However, abstractions typically assist programmers to determine software requirements, while also providing a portability path for existing applications designed and implemented with an abstraction layer in mind. Abstraction layers are a natural ingredient of the software development cycle exploited for fast deployment of new features. The use of abstraction layers might be minimized at later stages of development, if performance issues associated with them are determined.

Latencies in high performance networked systems are in many cases due to software overheads. The faster the networking hardware becomes, the more it becomes obvious that software dominates the communication latency. However, those overheads are not unrelated to the interface between the messaging layer software and the networking hardware. As a result, the latencies of the software could be reduced not only using software improvements, but also with the introduction of new hardware operations and features the software can take advantage of.

From the perspective of the application, the experienced performance does not depend solely on hardware performance, but also on the performance of the software. Some researchers propose designing networks and network interfaces in ways that either simplify or even replace software messaging layer functionality [6]. Operations and design choices that seem slow in the context of hardware might have the benefit of increasing software performance, so it is not immediately obvious which features should be supported by networking hardware. The existence of programmable hardware in the form of FPGAs assists the study of various possible hardware architectures and operations offered from the hardware to the software.

The Remote DMA (RDMA) feature has been around for many years. It was first proposed by architects in VAX clusters [22] and it is possible to find today a wealth of scientific and commercial hardware offering RDMA operations. InfiniBand [4] is an example of a specification implemented by networking hardware vendors, which includes memory communication semantics through the use of RDMA operations. The basic function performed by RDMA capable hardware is DMA across a network. RDMA makes possible for memory to be shared for communication purposes among processes across a network, with the network being anything from a traditional LAN or WAN to a NOC within a chip interconnecting multiple processors. RDMA offers the potential of eliminating copies between system and application buffers and has received interest in various contexts, including the Internet Engineering Task Force (IETF). An study of RDMA from the IETF can be found in [19].

As noted previously, MPI is an abstraction layer and it is a specification rather than an implementation. It is purposefully general enough to lend itself to implementations above a wide variety of underlying hardware architectures. MPI has been implemented over TCP/IP sockets and over shared memory architectures. An implementation of MPI over InfiniBand using RDMA operations can be found in [15]. A subset of MPI named Cell Messaging Layer has been implemented in multicore Cell [3][21] processor environments and is described in [17].

Our work focuses on the implementation of a subset of MPI in a CMP environment developed using an FPGA board. The processors in the CMP are interconnected through their Network Interfaces (NI's) to a Network On a Chip (NOC). The NI's and NOC offer various features to the software, including Remote Stores and RDMA operations. Descriptions of the platform and its features are provided in [5] [7] and [16]. We explore the minimum MPI functions that must be implemented in order to be able to run basic MPI applications over the specific platform. We also focus on analysing the performance of the MPI primitives we implement.

This master thesis is part of the integrated project Scalable Computer ARChitecture (SARC), which focuses on long-term research in advanced computer architecture. FORTH's responsibility for SARC project is the majority of the architectural research, some of the congestion control and all of the FPGA prototyping and NI development. This work aims to develop software that demostrates the capabilities of the underlying hardware for facilitating high performance communication among the processors within a CMP implemented using an FPGA board.

1.1 Thesis Contributions

In this thesis we present the software architecture and implementation of basic MPI primitives over a multicore processor hardware platform that offers RDMA operations.

We do not port an existing MPI implementation to the specific hardware platform due to limitations of the specific hardware platform. The limitations include lack of an OS and filesystem on the FPGA board, basic RISC-style processor functionality and small amount of local scratchpad memory to be used for RDMA operations and other tasks. We instead provide our own implementation of the basic MPI functions needed to run any meaningful MPI application.

Our MPI library uses RDMA Write operations, Remote Stores and Counters with associated notifications to satisfy the basic MPI specification requirements from the implementors of the MPI_Send() and MPI_Recv() functions. The MPI standard allows the MPI_Send() function to be implemented as non-blocking when using copies from application to system buffers. The MPI_Send() function of our implementation is blocking until a notification is provided by the hardware to the sender indicating that the application data of the sender have been delivered by the network to the scratchpad of the remote receiver. Our MPI_Send() function does not wait for the data to be copied from the remote scratchpad to the application buffer of the receiver.

Our library implementation includes the Eager and the Rendezvous Protocol that are typically used internally in MPI implementations to provide the semantics of various MPI communication modes. The Eager Protocol is typically used for small and the Rendezvous is used for large messages, while the choice between the two protocols is decided at run-time by the sender based on the application message size to be transferred to the remote side.

After measuring the latency of the Eager and Rendezvous Protocols, we have decided to set the threshold between the internal protocols to 48 bytes while the size of an eager buffer is set to 64 bytes. The choice was made with a focus on latency while also taking into account scratchpad memory size limitations.

The zero-byte 1-way latency of our implementation is 217 clock cycles. This latency has been computed as an average using a ping-pong test between 2 processes and is not representative of the actual latency of the individual MPI_Send() and MPI_Recv() functions. However, the ping-pong test is commonly used in the literature to measure the performance of MPI libraries and does provide an indication of an average expected latency.

The contributions of this master thesis are:

1. Design and implement basic blocking MPI primitives over a novel hardware architecture.
2. Implement both the Eager and the Rendezvous MPI library internal protocols.
3. Measure and analyse the latency components of the Eager and Rendezvous Protocols on the specific platform.
4. Use the performace results to set the threshold between the Eager and the Rendezvous Protocol as well as the eager buffer size.
5. Use the latency results to suggest ways of improving performance of the MPI library even further.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 refers to related work. Chapter 3 presents the software architecture of the MPI implementation over the hardware architecture of our platform. Chapter 4 presents MPI library implementation issues and experimental results. We summarize and draw conclusions in Chapter 5.

Chapter 2

Related Work

A significant amount of research and literature is available on the topic of MPI standard implementations over various types of underlying high performance interconnection networks and especially networks that offer RDMA operations. Each different hardware architecture typically offers a set of alternative communication operations to the software for developing a messaging layer and software developers need to decide which low level operation to use for each type of communication task. During the initial stage of developing an MPI library, the programmers choose the hardware features that can satisfy each MPI function requirement from the underlying layers. In cases where various alternatives exist, the performance and benefits (e.g. scalability) when using each alternative need to be evaluated.

Our work is an initial implementation of a basic subset of MPI over a Chip Multiproccesor (CMP) architecture that offers a plethora of communication mechanisms and services (such as RDMA Write, RDMA Read, Messages, Remote Stores, Counters and Queues). There exists no other MPI implementation on top of the same architecture to which we can directly compare the design and performance of our own MPI libary in equivalent conditions. However, the literature does offer evaluations of design alternatives that have been implemented in other hardware architectures with RDMA operations, such as networks based on InfiniBand Architecture (IBA) [4] and networks of the Cell Broadband Engine (CBE or Cell) processors [3][21]. We are using those implementations to describe the differences of our own implementation from other existing ones and to put our performance results into perspective.

In [9] an implementation of MPI over the native Verbs layer of InfiniBand is presented. InfiniBand supports both channel (send/receive) and memory semantics (RDMA). The VAPI interface [12] from Mellanox closely follows the Verbs semantics and the authors of this work use VAPI send/receive operations for the Eager Protocol data transfer. They also use VAPI send/receive operations for the exchange of control messages during a Rendezvous transfer. They chose to use RDMA Write operations for the actual data transfer of the Rendezvous Protocol. The evaluation of this work uses Mellanox's second generation adapter (InfiniHost) and switch (InfiniScale). Their implementation delivers around 9.5 usec latency for short messages on dual Intel Xeon 2.4 GHz systems with PCI-X 64-bit 133 MHz interfaces, while the Verbs layer of the InfiniHost adapter delivers 6.9 usec latency for small messages. The hardware architecture used to obtain those performance results is quite different from ours. A rough conclusion may be that their messaging library adds approximately 2.6 usec to the short message latency of VAPI, which is slightly above 27% of the total short message latency using a high performance processor. Their MPI library implementation is also different from ours, since our hardware platform does not offer send/receive operations. Our library uses RDMA Write operations for both Eager and Rendezvous Protocols. We also use the notification mechanism of our platform for the exchange of the FIN control message of the Rendezvous Protocol.

[11] continues the work described in the previous paragraph and tries to bring the benefit of RDMA operations to the small messages of the Eager protocol and to the control messages of the Rendezvous Protocol. RDMA Write is still used for the Rendezvous Data. They focus on RDMA Write because RDMA Write usually had better performance than RDMA Read in their hardware at the time of writing. They use send/receive when it is not possible to perform RDMA Write for small and control messages. Our MPI library does not use send/receive, as said previously. However, our design is very similar to this work in many respects. We have used fixed-size, preallocated buffers for Eager Protocol at both the sender and receiver sides as they did. We also have a persistent buffer association between the sender and receiver sides and the buffers are organized as rings. The detection of incoming eager messages at the receiver side in their work is handled by introducing multiple flags within an eager buffer, while also relying on the hardware to write bytes in order at the destination buffer because InfiniBand implementations support it, although not specified in the InfiniBand standard. Our MPI library does not introduce multiple flags. We cannot rely on our hardware to write bytes in order at the destination. We exploit the notification mechanism of our platform to set the ready flag in an eager receive buffer. In our design the ready flag can be placed at the header or trailer of an eager buffer and we have chosen to place it at the address of the eager buffer itself. Our MPI library never sets the ready flag within an eager buffer. Only the hardware sets this field on RDMA Write completion. The software at the receiver resets the flag after the eager buffer has been consumed. Their implementation delivers around 6.8 usec latency for short messages on dual Intel Xeon 2.4 GHz systems with PCI-X 64-bit 133 MHz interfaces.

The RDMA Channel interface abstraction of MPICH2 is used to provide an MPI implementation over InfiniBand in the work described in [10]. This implementation delivers around 7.6 usec latency for short messages on dual Intel Xeon 2.4 GHz systems with PCI-X 64-bit 133 MHz interfaces and demonstartes that the RDMA Channel abstraction of MPICH2 can deliver reasonable performance with reduced development effort. Our implementation does not port an existing MPI implementation to our hardware by using an abstraction with only a few functions to implement. We instead provide a new implementation targeted to our hardware platform and our initial release supports only a limited subset of MPI features. Our platform is an FPGA prototype and our resources do not allow porting of an existing implementation. In the design described in [10], small messages are transferred using RDMA Write, while for large messages RDMA Read (instead of RDMA Write) is used for the Rendezvous data transfer. Our implementation does not exploit the RDMA Read support of our platform. We chose to use RDMA Write in our initial implementation to provide a baseline for MPI library performance. Future work will aim to use RDMA Read for the Rendezvous data transfer. As described in [20], the use of RDMA Read reduces the number of control messages in the Rendezvous Protocol and allows the receiver to proceed independently of the sender after the initial Rendezvous control message is sent, with the result being better overlap of communication with computation. We conclude from those implementations that the use of a hardware operation that might seem slower at the hardware level (RDMA Read compared to RDMA Write) might result in better overall performance, because of a reduction in software overheads and synchronization points between the sender and receiver. Similar concepts about features of networks and network interfaces that simplify messaging layer software have been described in [6].

In [17] an MPI implementation is developed for the Cell. The implementation does not support the Eager Protocol and provides only an equivalent of the Rendezvous using RDMA PUT operations. The authors propose receiver-initiated message passing, in which the receiver initiates the Rendezvous Protocol. We agree that the support of MPI_ANY_SOURCE is non-trivial with receiver-initiated message passing. Their implementation does not support MPI_ANY_SOURCE and neither does ours, since we provide a new implementation in an environment with limited resources. However, an additional reason exists for sender-initiated message passing not being extensively used in the MPI literature. MPI allows the application message size of the sender to be different from the application message size specified by the receiver and the minimum of the two should be tranferred. MPI applications might use such a feature to post general purpose application buffers for receiving messages from either any source or having any tag or for any other reason. As described in [20], only the sender side knows the size of the actual data to be transferred and can make an efficient decision to use either the Eager or the Rendezvous Protocol. [cellmpi] does not implement the Eager Protocol, so their implementation does not need to make a choice between protocols. Our implementation supports both protocols and we therefore have chosen sender-initiated message passing. Another reason for receiver-initiated message passing not being commonly utilized is related to how the Rendezvous Protocol is implemented using RDMA Read in the literature. As described in [20], sender-initiated message passing with RDMA Read has the same number of control messages as receiver-initiated message passing, the number of synchronization points between the sender and receiver is also reduced, and both the Eager and Rendezvous Protocol can be supported in a manner efficient for the application. The implementation of MPI for the Cell described in [17] has a zero-byte latency of 0.272 usec or 870 Synergistic Processor Element (SPE) clocks within the same Cell, with the SPE clock operating at 3.2 Ghz. Our implementation has a zero-byte latency of 217 clock cycles with the clock of our FPGA prototype being approximately 75 Mhz, which yields a zero-byte latency of 2.893 usec. The two hardware architectures are obviously quite different.

MPI has also been implemented in [18] across multiple FPGAs. The TMD-MPI design is minimal due to similar limitations we face on our own FPGA prototype. Their testbed system has two networks with the one being used for intra-FPGA communication and the other for inter-FPGA communication. They support only the Rendezvous Protocol and their hardware does not offer RDMA operations. The on-chip communication uses internal BRAMs. A single BRAM (64KB) contains code and data. The testbed is running at 40 Mhz and has an on-chip zero-byte latency of 17usec or 680 cycles at 40 Mhz.

Chapter 3

Hardware and Software Architecture

This chapter provides a detailed description of the software architecture of the MPI library. Since the MPI library interfaces directly to the hardware, we provide a high-level description of the hardware architecture. We also examine how the software requirements can be mapped into hardware operations. The hardware is described from the perspective of the software programmer and includes only the details related to software design and implementation.

3.1 Hardware Architecture Overview

This section provides an introduction to the hardware architecture of our system. We do not provide a detailed description of the hardware. Additional information about the hardware will be provided, when needed, at later sections and as we describe software design and implementation. A detailed description of the hardware can be found in [5] [16].

3.1.1 Hardware Platform Abstract Architecture

Our platform contains 4 processors interconnected by a NOC. An abstract view of the entire CMP is provided in Figure 3.1, while a high-level diagram of each processor block is provided in Figure 3.2. A 256MB DDR2 SDRAM is also connected to the NOC and is accessible from all the processors. Each of the 4 processors has an L1 and an L2 data cache. The size of each L1 data cache is 4 KB and the size of each L2 data cache is 64 KB. Part of each L2 data cache can be configured by software at run-time to operate as scratchpad. The configuration is as simple as writing a specific value to the tags of each cache line we desire to have it operate as scratchpad instead of cache.

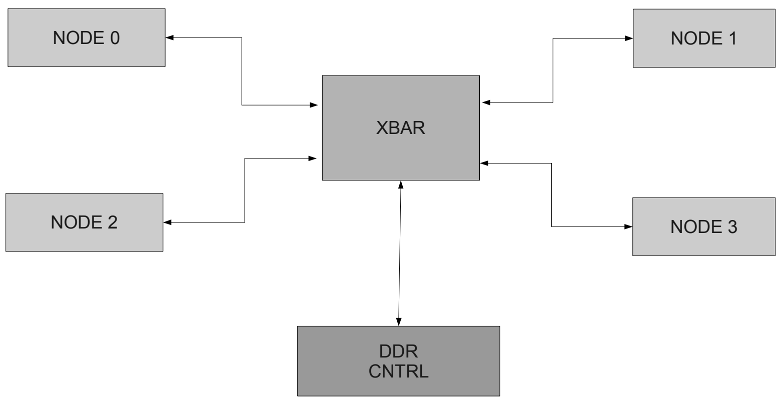

Figure 3.1: Abstract diagram of CMP with 4 processors

From a programmer's perspective, a scratchpad region within the L2 data cache is nothing more than a memory close to the processor, and therefore fast, that can be directly addressed like any other memory. The hardware will not evict any of the cache lines that have been configured by the software to operate as scratchpad.

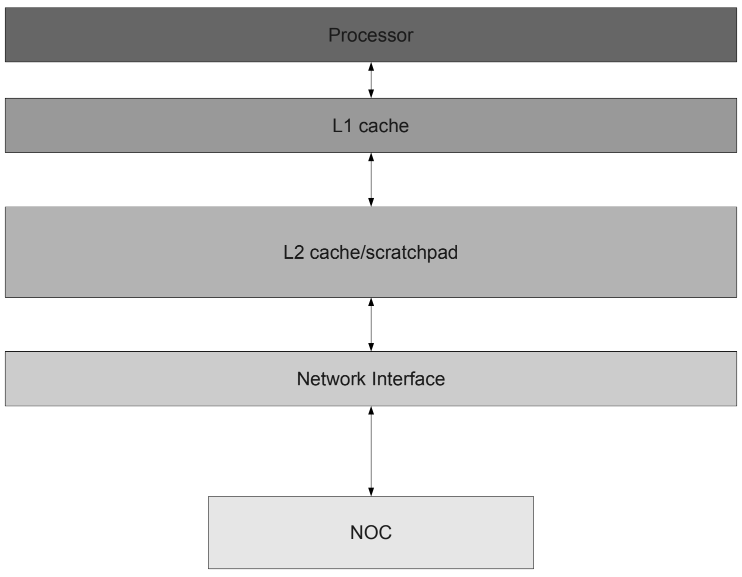

Figure 3.2: CMP Node

Each L2 data cache is 4-way associative and we must refrain from allocating all the available cache ways as scratchpad, because we must have at least some data cache for program data that are not explicitly mapped into scratchpad regions. Taking this restriction into account as well as the L2 cache size of 64KB, it becomes obvious we should carefully choose the software data to allocate into scratchpad regions.

We should also keep in mind the trade-off between speed of access to scratchpad region data and possible frequent eviction from the cache of other cacheable data of our program. The more scratchpad regions we allocate, the less cache space remains for cacheable data, so the data in scratchpad should generally be more frequently used than the remaining data that are left to be cached.

The previous requirement seems to introduce a slight paradox regarding expected cache behavior. We typically expect cached data to exhibit spatial or temporal locality. If the expectation of infrequent use of cached data is fully realized, then the performance of the cache might deteriorate to the point of being incapable to hide the DRAM latency. However, a low locality behavior cannot be handled by any cache scheme. Frequent eviction due to reduced cache size is a greater concern. In the average case we expect the cached data to be used often, but the data in scratchpad to be used more often than those that are cached. The point is that we proactively cache data we expect to be used often and this practice, although manual, has potential of achieving better code performance. We still need to carefully choose the data to place into scratchpad regions though.

3.1.2 Hardware Operations

Each L2 data cache of our platform is integrated with an NI. Communication between nodes is achieved via the NI's, which interconnect each processor to the NOC. Besides being able to allocate general software data into scratchpad regions, we also need to allocate data into scratchpad for the purposes of issuing commands to the NI and getting responses from the NI. If notifications about completion of operations are also desired, than we also need to allocate Counter objects within scratchpad regions. The logical view of the L2-cache run-time configurability is shown in Figure 3.3.

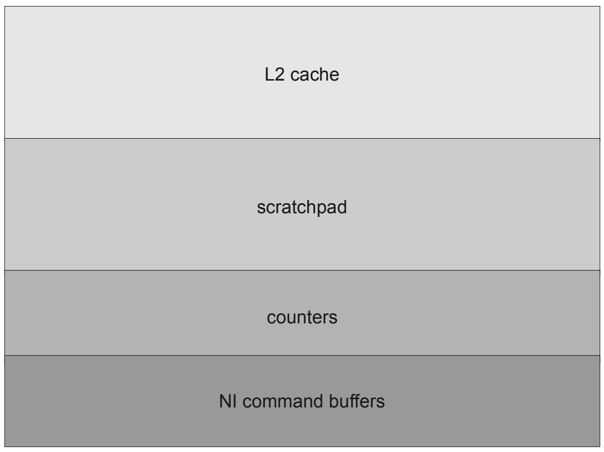

Figure 3.3: L2 cache logical run-time configurability

Each scratchpad memory is identified by using a specific memory address range that is parametrized by the CPU_ID of the CPU physically closest to a specific scratchpad memory. We will typically say that each CPU has a scratchpad memory, although it should be noted that each scratchpad memory is accesible by other processors and ownership of scratchpad is not strictly assumed.

In the next paragraphs we describe the operations that are offered from the hardware to the software for the purposes of message exchange together with the conventions that must be followed when software communicates with the NI's. The hardware features offered are the following:

Counters/Notifications

A Counter is an object allocated within special scratchpad lines by writing a specific value into the cache tag of the line. A Counter is typically initialized with a 24-bit signed value representing the size of the Counter. When issuing RDMA operations a Counter object is used to detect completion of the RDMA operations. The value of the Counter in this case is associated with the size of the RDMA operation. By convention, if the size of the RDMA operation is dma_size, then software should provide the value (0 – dma_size) to the Counter instead of the value dma_size. This convention benefits hardware design and is trivially implemented in software. The value of the Counter is written at the address of the Counter itself. Software can also provide up to 4 scratchpad memory addresses that should be notified about subsequent operation completion. Those addresses are written at the second half of the Counter scratchpad line and can be either local or remote addresses. Counter configuration alternatives are shown in figure Figure 3.4. More information about how exactly the notification addresses can be used is provided in the description of RDMA operations that follows.

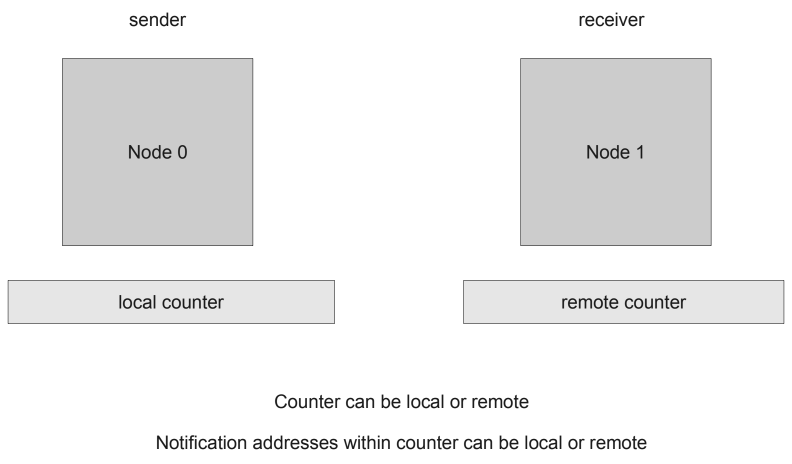
sender
receiver
Node 0
Node 1
local counter
remote counter
Counter can be local or remote
Notification addresses within counter can be local or remote

Figure 3.4: Counter configuration alternatives

RDMA Write/Read

Software can instruct the NI to perform RDMA operations. The typical software steps are as follows:

a. Software determines the source and destination memory addresses associated with the RDMA operation. These memory addresses must belong to a scratchpad region. Any processor can issue an RDMA operation with the memory addresses being in any scratchpad region of any scratchpad in the platform (i.e. any combination of remote or local scratchpad addresses is allowed).

b. Software determines whether notification(s) about completion of an RDMA operation should be provided by the NI. If this is so, then software allocates a Counter object in scratchpad region and updates the Counter with the desired notification addresses. The notification addresses must also be located in scratchpad regions for the NI to be able to update them after operation completion. When the operation is complete, NI will write the value 1 into the notification addresses, so software must set the contents of any notification address to a value other than 1 before the RDMA operation is initiated to be able to detect completion at a later time via polling. Note also that the value of the Counter is set to (0 – dma_size). As shown in Figure 3.4, a Counter does not need to be local to the processor issuing an RDMA operation.

c. Software allocates an NI command buffer in scratchpad region if one has not already been allocated and issues a sequence of stores into the NI buffer supplying the RDMA operation arguments. The NI automatically detects command buffer completion and proceeds to perform requested operation without any further software intervention.

d. Software can be informed about progress of an RDMA operation by either testing for the value of 1 at any notification addresses previously supplied or by testing the first location of an NI command buffer for the value of 0. The later method does not mean that the operation has been completed, but is useful in order for software to determine whether an NI command buffer can be reused.

Remote Store

Each processor can issue a store to a remote scratchpad address. From the perspective of the software, the remote store is implemented like any other store. Typically a pointer holds the value of the remote address to write data to and the remote store is as simple as writing the contents of the pointer via a usual assignment of a value to the dereferenced pointer.

Messages

A Message is a compromise between a Remote Store and an RDMA Write. Issuing a Message transfer is similar to the issuing of an RDMA command to the NI. A Message can be used to write up to 5 4-byte words to a scratchpad address, while a store can write up to 4 bytes to an address. A Remote store has lower initiation overhead, since the initiation of a Message transfer requires itself a sequence of stores.

Queues

The hardware provides mechanisms for the software to define Queues in scratchpad regions that can be partly controlled and updated by the hardware and partly by software. It was not clear how we could incorporate them into our MPI library implementation. Besides, at the time of development of the library a new version of the hardware was being deployed and the Queues had not been tested. We therefore did not use them in our library implementation. More information about the Queues offered by the hardware can be found in [5].

3.1.3 Mapping Hardware Operations to Software Requirements

The Remote Store can be used for small data transfers between processors. Any control messages used by our library during MPI application data transfers are candidates for usage of Remote Stores. A performance gain from use of a Remote Store is expected when it is used with messages for which we can infer the completion by the data actually stored by the operation or when we can wait for the data at any point in time (i.e. we only care for the data to eventually arrive at some point and do not demand to consume the latest such data).

A Message can be used for data transfers up to 5 4-byte words. Control messages are candidates for using a Message if a Remote store to a single 4-byte word is not sufficient. The performance of 5 stores compared to the performance of a single Message should be explored.

Any typical data transfer can be impelemented with RDMA operations. RDMA operations do have an initiation overhead, so, from a performance perspective, the point over which an RDMA operation is more beneficial than a Remote Store or Message should be explored.

Counters and their associated notifications will be used to detect completion of RDMA operations in order to determine whether normal buffers or NI command buffers or the Counters themselves found in scratchpad regions can be reused.

3.2 MPI Library Programming Environment

This section includes some implementation issues because they directly affect the software design choices. We briefly mention them in this specific chapter rather than delaying them until Chapter 4.

The Application Programming Interface (API) described within the MPI specification is typically implemented using the C programming language, although implementations using other languages such as FORTRAN or even combinations of programming languages are also possible. Our implementation is based solely on the C programming language.

The FPGA prototyping environment we use to implement and test our software does not include an operating system. We only have a C runtime system with minimal C library support. We do not have a filesystem on the board either. The 4 processors in our system are Xilinx MicroBlaze processors and we use the mb-gcc compiler to compile our programs with the assistance of Xilinx tools that facilitate software development.

The platform includes a DDR SDRAM. The program text of each processor is assigned to different non-overlapping DRAM regions with the assistance of linker scripts specifically developed for this purpose. Programs are loaded at run-time from the DRAM into the instruction caches of the processors.

It is hardly feasible to port any known MPI implementation into our system, since our DRAM is limited to 256 MB and we do not even have an OS. Consequently, our implementation begins from scratch and we focus on providing a subset of MPI functions with the goal of extending the software code base of our hardware platform. Our focus is on basic functionality instead of providing a full MPI standard implementation. We explore the practical issues faced when trying to implement the basic MPI functions in a multicore FPGA environment. Such issues have already been explored by other researchers during the development of the Cell Messaging Layer in the context of Cell processor environments. The Cell Messaging Layer does not implement non-blocking MPI functions and we chose not to support those functions either in our initial MPI implementation.

The MPI standard purposefully does not constrain MPI library implementations with respect to various software design choices. Still, basic MPI primitives do have specific requirements from the underlying software layers. Our library implementation is ultimately limited by the available hardware resources of the specific FGPA prototype used to implement and test our software, so practical compromises must be made. We do not claim to strictly follow the standard, but we strive to fundamentally comply to usual expectations from MPI application programmers.

3.3 MPI Application General Structure

An MPI program consists of independent processes executing their own instructions and generally each process has its own address space. In order for two or more processes to exchange information, explicit messages must be sent from one processes to another. MPI employs the concept of a communicator to capture the abstraction of a group of processes capable of exchanging messages for a specific purpose. A typical MPI application developed using the C programming language exhibits the following general source code structure:

```
int main(void)
{
        MPI_Init();

        MPI_Comm_size(MPI_COMM_WORLD, &size);

        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        if (0 == my_rank)
        {

        // Computation

                                MPI_Send();

        // Computation

                                MPI_Recv();
        }

        MPI_Finalize();
}
```

The typical MPI application source code begins with a function call that initializes the MPI library and ends with a function call that terminates the library environment. A process rank is an identifier used to refer to each process and select the code that each process should run. The calls to the MPI library are embedded in normal C source code and are no different than any other typical library function call.

As stated previously, our library design begins from scratch and our main goal is to extend the software code base of our hardware platform. We implement only a subset of the MPI functions and focus on basic functionality instead of providing a full MPI standard implementation. We explore the practical issues faced when trying to implement the basic MPI functions in a multicore FPGA environment. The section that follows provides detailed descriptions of the subset of MPI functions we chose to implement. We determine the requirements of applications from the MPI library and explore how these requirements can be satisfied in our specific environment.

3.4 Description of MPI Primitives

The MPI primitives we implement are described in the following paragraphs. These primitives are absolutely required in order to run any meaningful MPI program. The fact that we do not implement non-blocking MPI functions at this stage has an important consequence for MPI applications run on top of our library: all message exchanges between processes should be carefully arranged to avoid deadlocks. In other words, the MPI application programmer cannot assume that deadlock situations would be resolved in any implicit way by the MPI library.

3.4.1 MPI_Init() and MPI_Finalize()

All MPI programs must begin with a call to MPI_Init() and end with a call to MPI_Finalize(). The call to MPI_Init() prepares the necessary data structures for subsequent calls into the MPI library, while the call to MPI_Finalize() cleans up and terminates the library environment. Any calls into the MPI library before the invocation of MPI_Init() or after the invocation of MPI_Finalize() are expected to be erroneous and such practice is therefore not recommended to MPI application programmers.

The work performed by the MPI_Init() function is closely related to the implementation of MPI_Send() and MPI_Recv() functions. The latter functions, when implemented using RDMA operations with memory semantics, typically require memory address conventions for message exchange. Those conventions are partly realized during library initialization. The correctness of the MPI_Send() and MPI_Recv() functions highly depends on the correctness of the initialization.

Of particular importance is the requirement for synchronization between processes of the system before the MPI_Init() function returns at each processes. At this point, every other processes must have been initialized, otherwise programs might be erroneous in a non-deterministic way or might simply crash. Synchronization among processes at the end of MPI_Init() must be confirmed whenever an issue that manifests itself in a non-deterministic way appears during the operation of the library and should be kept in mind whenever changes need to be made to any of the initialization routines. Alternatively, applications could explicitly call MPI_Barrier() after MPI_Init() to avoid initialization related errors.

3.4.2 MPI_Comm_size() and MPI_Comm_rank()

The utility functions MPI_Comm_size() and MPI_Comm_rank() are typically used by an MPI application immediately after the MPI library has been initialized by invocation of MPI_Init(). A typical MPI application uses the rank parameter and size of communicator to distribute unique work among processes.

Through a call to MPI_Comm_size() each processess obtains the size of the communicator MPI_COMM_WORLD, which is also the only communicator supported by our implementation. The size of MPI_COMM_WORLD is fundamentally the number of processes running in our system. We support a single process per available processor. Since 4 processor exist in our multicore environment, the size returned by a call to MPI_Comm_size() is also 4.

Through a call to MPI_Comm_rank() each process obtains its rank within the MPI_COMM_WORLD communicator. Generally, in an MPI environment, each process is uniquelly identified by a rank within a communicator. Since MPI_COMM_WORLD communicator is the only communicator supported by our implementation, the rank suffices to uniquely identify a process in our platform. Each processor in our system already has a unique CPU_ID. We can therefore perform a one-to-one mapping from the processor CPU_ID into the MPI rank of each running process. Consequently, CPU_ID and MPI rank can be considered equivalent for the purposes of our implementation and from now on we can use any of those terms to identify a process, depending on the context.

3.4.3 MPI_Barrier()

A barrier forces all processes to wait until all processes reach the barrier and afterwards releases all the processes. Implementation of a barrier for synchronization among processes in our system is important for the correctness of MPI_Init() and therefore the correctness of any MPI application that uses our library. Our existing system software includes an implementation of barriers. The algorithm used is a sense-reversing barrier. A sense-reversing barrier avoids undesired interactions between barrier instances, and particularly situations where a process races ahead and gets to the barrier again before the last process has left. More information on sense-reversing barriers can be found in [5]. We simply comply to the requirements of the system software library and follow the conventions for the practical definition of such a barrier.

The only communicator we implement is MPI_COMM_WORLD. This communicator is absolutely necessary for any MPI implementation and is used to capture the abstraction of a point-to-point message path between each pair of processes existing in our system. A barrier call that only uses the MPI_COMM_WORLD communicator practically means that all processes in our system must synchronize at each occurence of MPI_Barrier() function call in a program. Note that the implementors of the Cell Messaging Layer have also chosen to implement this communicator only. This implementation choice is reasonable in a multicore environment with increased conserns about high speed memory availability and simultaneous demands for high performance.

3.4.4 MPI_Send()

The MPI_Send() function initiates a point-to-point message transfer from the process that calls the function towards a second process. The MPI_Send() function is referred to as a standard send with blocking semantics. The syntax of the blocking send operation is the following:

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

The MPI_Send() function does not alter any of the arguments supplied in an invocation of the function. The *buf* is the address of the application send buffer, *count* is the number of elements contained in *buf*, *datatype* is the datatype of each element in buf, *dest* is the rank of the destination process, *tag* is an extra field added to an outgoing message (used in application specific ways to differentiate among messages), and *comm* is a communicator object.

The *rank* and *comm* uniquely identify the receiving process and, consequently, a standard send can only initiate point-to-point message transfers between a sending and a receiving processes. In our system, the MPI rank of the destination process needs to be specified in the parameters passed to the function. The communicator parameter in our system must always be MPI_COMM_WORLD. Support for MPI_COMM_WORLD is required, while it is typical to be the only communicator supported in initial MPI library implementations in multicore environments with limited resources.

Our implementation will not support tags. The use of tags complicates the message matching logic at the receiver side and is not supported in other similar implementations. More information about the matching logic is provided when describing the MPI_Recv() function.

When MPI_Send() returns to the calling environment, the application assumes that the application buffer provided when the standard send operation was issued can safely be reused. The MPI library implementation can either deliver the application message to the matching receive buffer or use system buffers to temporarily store a pending message. So, when this function returns, it is not generally true that the application message has been necessarily delivered to the remote receiving application side. On return from the function, an application message might only have been copied to system buffers and the NI instructed to deliver it to the remote side. Still, from the perspective of the MPI application the send buffer can be reused and this is all that matters for all practical application programming purposes. Such a call is characterized as being non-local, because successful completion might depend on the occurence of a matching receive.

Message buffering requires additional memory to be allocated either at startup or progressively during the lifetime of a program with potential complexity about how to return such memory to the system when it is no longer needed. Another disadvantage is the resulting copy overhead between application and sytem buffers at the sending and receiving processes. However, the potential benefit is a decoupling of the sender from the receiver, which could improve overlap of communication with computation in some cases on the sender side. The use of system buffering is yet another instance of the engineering principle that sometimes trades memory for speed.

Additional reasons to use system buffering and copies between application and system buffers appear in environments offering RDMA operations: not all memory addresses can be used for RDMA, some overhead occurs during allocation of RDMA-capable memory, and size of RDMA-capable memory is usually limited or has specific alignment restrictions when used. Even if all memory can be used for RDMA with byte-level accuracy, lack of support for cache coherence might introduce errors in programs. Our system does not yet support cache coherence and, therefore, the support for system buffering in our implementation seems to be needed at least at an initial stage. Such support is realized via the implementation of the Eager Protocol, with details being described in later sections.

Zero-copy support in the MPI libraries is typically desired in order to avoid the known overheads of copies from application to system buffers, especially for messages perceived as large. Such support is realized via the implementation of the Rendezvous Protocol, with more details provided in later sections. Support for zero-copy implies that the starting address of the application buffer must be part of an RDMA-capable region. The application buffer must therefore have been allocated in such a region from the beginning or at least before the address of the buffer is handed off to the MPI library. A typical technique to address this requirement is to provide hooks into the malloc() function. The RDMA-capable memory is not infinite and not all memory allocated by an application will be used for communication purposes. The allocation of RDMA-capable memory for computation purposes only might be a disadvantage in some cases. Tradeoffs are encountered in both the Eager and the Rendezvous Protocol implementations, same way tradeoffs appear in most engineering choices, so, balance and tuning will typically be needed to get the best of both worlds.

3.4.5 MPI_Recv()

The MPI_Recv() function delivers a message to a process. Typically another processes has either previously initiated a point-to-point message transfer towards the process that receives the message or is expected to do so in the near future (i.e. data might not be yet ready when the receive is posted). The MPI_Recv() function is referred to as a standard receive with blocking semantics. The syntax of the blocking receive operation is the following:

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,*
*MPI_Comm comm, MPI_Status *status)*

Most of the arguments of an MPI_Recv() and an MPI_Send() function call have similar meanings, with the main difference now being that arguments are interpreted from the prespective of the receiver (i.e. *buf* now becomes the initial address of the receive buffer, and *rank* is the rank of the sending processes). There exist some differences as far as the *count* and *source* arguments are concerned, while *status* is an additional argument we do not encounter in a standard send.

The argument *count* supplies the length of the received message and must be less than or equal to the length of the receive buffer. According to the MPI standard, if a message shorter than the receive buffer arrives, then only the memory locations corresponding to the shorter message will be modified. If a message larger than the receive buffer arrives, then an overflow error might be returned, but still, quality implementations must ensure that memory locations beyond the boundaries of the receive buffer will not be modified. In conlusion, the length of the message written to the receive buffer is the shortest among the sender and receive buffer length. We cannot assume that the lengths of the send and correponding receive buffers will be identical. MPI applications sometimes allocate a large buffer and use it to receive messages of various sizes. If a blocking receive call is used to receive messages, then a flexible receive buffer size allows a program to avoid deadlocks when matching the size of the transferred message.

A receive operation must select one among potentially many messages delivered to the receiving processes at the time the receive is posted. This selection is determined by the value of the message envelope. Specifically, if envelope matches the *source*, *tag* and *comm* values specified by the receive operation, then the message is a candidate for reception by the application.

When a receive operation is posted, multiple messages that have already arrived might match the operation and the MPI library must decide which one of those messages must be delivered to the MPI application. If there are multiple matching messages, then, on reception, the MPI library must maintain the exact order the messages were sent by the sending process. MPI terminology refers to this requirement by stating that messages are non-overtaking and implies the existence of a virtual FIFO between each sender and each receiver.

According to the MPI specification, a receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, to indicate that any source and/or tag can be accepted by the application. The use of wildcards modifies the matching logic and a message can be received only if it is addressed to the receiving process, has a matching communicator, has matching source unless source= MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag= MPI_ANY_TAG in the pattern. The use of wildcards complicates the matching logic and introduces non-determinism in MPI libraries.

We do not offer support for MPI_ANY_SOURCE and MPI_ANY_TAG in our implementation. We do not support tags in general. The argument *status,* which is used to provide additional information about reception to the appluication is not supported. Note that this argument is typically used by an application that uses wildcards to determine the *source* and *tag* fields of a received message. Since our implementation does not support wildcards, it is not particularly meaningful to support the *status* argument. Note also that the Cell Messaging Layer has followed similar implementation choices. In later sections we will provide more details about how the requirement for a virtual FIFO between each sender and each receiver is realized in our implementation.

3.5 Eager and Rendezvous Protocols

Two internal protocols are typically used in MPI library implementations to provide message exchange between processes: Eager and Rendezvous. Both protocols are fundamentally used to facilitate a data transfer, but differ in various respects. Each protocol has advantages and disadvantages and the choice between them usually depends on the size of the application message being transferred. A detailed desctiption of Eager and Rendezvous Protocols is provided in the paragraphs that follow.

3.5.1 Eager Protocol

When Eager Protocol is employed, sender eager writes data to the receiver side without any apparent negotiation. Eager Protocol is depicted in Figure 3.5. As noted in [12], Eager Protocol uses system buffer preallocation at the receiver in order for the sender to be able to write data directly into the remote memory space, while in Rendezvous the sender is informed by the receiver about remote available buffers.

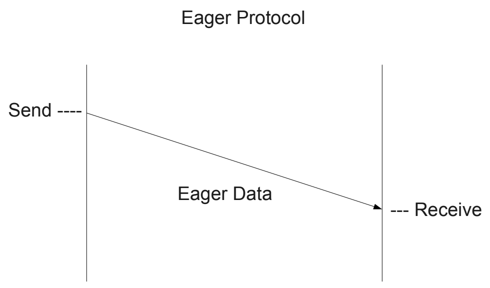

Figure 3.5: Eager Protocol

The Eager Protocol approach also utilizes memory copies between application and system buffers at the sender and receiver sides as shown in Figure 3.7.

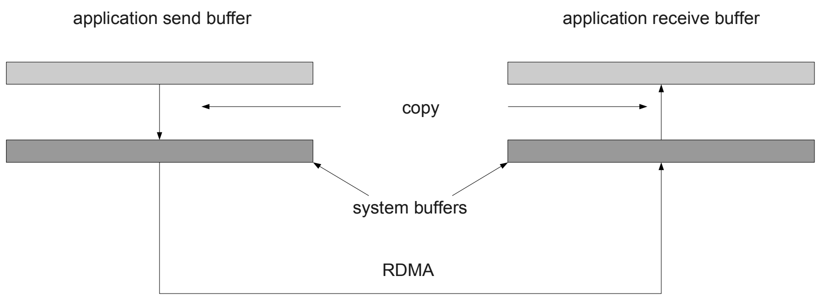

Figure 3.6: Copies between application and system buffers

It is obvious that the Eager approach suffers from scalability issues due to system buffer memory preallocation, and more so if per-processes memory preallocation occurs. Eager protocol fundamentally trades memory for simplicity and speed. This trading typically occurs only for smaller messages to avoid memory exhaustion. For small messages it might faster to copy the message into system buffers than to suffer the software steps overhead of the Rendezvous Protocol. It is expected that usage of Eager protocol for message exchange will become prohibitive in the case of larger messages, not only due to memory availability concerns, but also for performance reasons associated with the copy overhead, even if the necessary system buffer memory had not been an issue.

The choice between Eager and Rendezvous Protocols depends on the size of the MPI application message being transferred and is decided at run-time. The Eager-to-Rendezvous message size threshold is a compile-time design parameter of our software library and needs to be tuned and adjusted to the specific hardware environment of our system.

3.5.2 Rendezvous Protocol

The steps of the Rendezvous Protocol are shown in Figure 3.7. We implement the Rendezvous Protocol using RDMA Write operations. The RDMA Write based Rendezvous protocol generally uses 3 control messages for a single message transfer between 2 processes. As shown in Figure 3.7, ender queries the receiver side about the remote memory address to write data to (Request To Send, RTS), receiver responds with the requested memory address (Clear To Send, CTS), sender completes the data transfer (Rendezvous Data) and finally sender informs the receiver about completion of the operation (Rendezvous Finish, FIN).

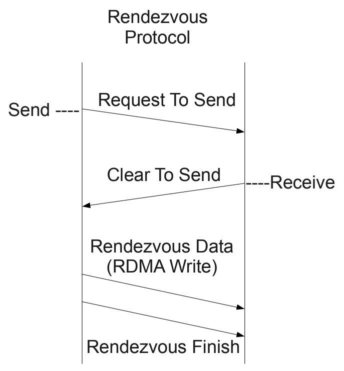

Figure 3.7: RDMA Write based Rendezvous Protocol

There exist various techniques to reduce the number of software control messages associated with the Rendezvous Protocol. The Rendezvous Finish control message could be handled automatically by hardware notification mechanisms after the Rendezvous Data arrive at the receiver. Another technique uses RDMA Read operations for the Rendezvous Data transfer instead of RDMA Write. Note that this technique modifies the protocol steps slightly as shown in Figure 3.8.

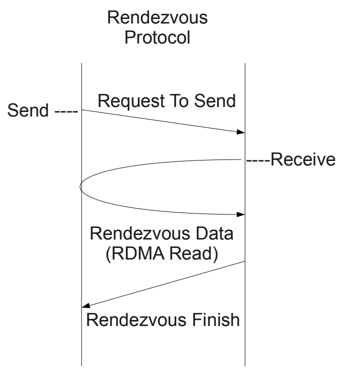

Figure 3.8: RDMA Read based Rendezvous Protocol

Chapter 4

Library Implementation and Performance

This section describes a software library implementation of blocking MPI primitives over a chip multiprocessor environment developed in an FPGA. We exploit enhanced features offered by the hardware for the purposes of low-latency communication and messaging layer software development. Our implementation takes into account the limited fast memory resources, software utility availability issues and functionality limitations typically encountered in FPGA platforms.

4.1 FPGA Prototyping Environment

4.1.1 Target FPGA

The hardware prototype of our system has been implemented in a Xilinx Virtex-5 FPGA. The chip multiprocessor contains 4 MicroBlaze soft-cores as processors. The processors are RISC-style processors, 32-bit, in-order, and have been configured to include a 5-stage pipeline for performance reasons. High performace multiplication and division is not supported by the hardware, so software should refrain from using such operations to avoid a performance penalty.

Each processor has a private L1 data cache and a configurable L2 cache/scratchpad memory tightly coupled with an NI. Instruction caches have been activated. The prototype includes a 256 Mbyte DDR2 SDRAM to be shared among processors and be used as main memory. Communication among processors and the off-chip DDR memory controller is achieved via a 32-bit, 5-port input-queued crossbar switch with round-robin scheduling policy.

The operating clock frequency of the system is approximately 75 MHz due to DDR controller issues and the complexity of the hardware design. Our performance measurements will be provided in units of clock cycles instead of seconds. Note that most MPI implementations report performance numbers in units of seconds.

4.1.2 Software Development Environment

As was described in the previous chapter, our software design has been affected by the fact that we do not have an OS or filesystem on the FPGA board. We only have a C run-time environment with limited C library support. We developed our code and downloaded it to the FPGA using Xilinx development tools. The compiler we used is mb-gcc and we compiled our programs with optimization level -O3.

The mapping of program sections to the memories of our system has been achieved with the use of linker scripts. The 256 Mbyte DRAM of our system has been equally partitioned among the 4 processors. We generate 4 executables and each processor has its own address space. The program .text is mapped to the last 16MB of each partition. The other required sections, including the stack and heap, of each executable are mapped to the first 48MB of each DRAM partition.

4.2 Messaging Layer Components

4.2.1 Data Structures

Our implementation of Eager Protocol uses RDMA Write to eagerly copy application data towards the receiver application buffer using intermediate copies to system buffers. A sender that initiates a message transfer needs to know beforehand the remote memory address of the receiver system buffer in order to instruct the NI to perform the RDMA operation. Consequently, the system buffers must be preallocated and a convention between each pair of sender and receiver should be maintained in order for each sender to know at any given point in time the remote address where the next data are expected by the receiver.

The system eager buffer memory space used for copying user data and afterwards initiating RDMA operations is fixed size and obtained at startup. We currently obtain 1 way of the L2 cache for the purposes of our library. After we obtain the RDMA memory space, we use it to allocate queues for all existing processes. The number of processes is equal to the number of processors in the system. The maximum number of CMP nodes connected to the prototyping platform is 4.

We allocate a send queue and a corresponding receive queue for each pair of processes. Sender 0 uses send queue 1 to send data to receiver 1 while receiver 1 uses receive queue 0 to receive data from sender 0, and so on. We maintain a mapping between a send queue and its corresponding receive queue. Each of those mappings can be considered as a point-to-point unidirectional virtual circuit between a sending and a receiving process.

The system buffers in each send queue are persistently associated with their corresponding system buffers in a receive queue. Persistent buffer association is shown in Figure 4.1, has been used in other RDMA based Eager protocol implementations and eases coordination of buffers and flow control between a sender and a receiver.

The send and receive queue buffers are organized as rings with corresponding head and tail pointers. A queue is empty if the pointers are equal and full if an advance of the tail pointer by one (mod N, where N is the number of buffers) would cause the tail to reach the value of the head.

The sender writes data at the tail position of a send queue and then copies them at the corresponding position of the remote receive queue using RDMA Write. Therefore, the sender always knows the tail position of the send and corresponding receive queue. The sender does not know the status of the head, since only the receiver acts on this variable. Each sender needs to monitor the head and tail of a send/receive queue pair in order to detect a queue full condition. Note that in the scheme we describe, and from the perspective of the receiver, only the receive head pointer is significant. The main responsibility for flow control is handed off to the sender.

The receiver can inform the sender about the value of the queue pair head by writing the head value to a specific space within the scratchpad memory of the sender. We have allocated control space in scratchpad memory at each sender for each remote receiver. Each receiver can write the value of the queue pair head to this control space using a Remote Store. In conclusion, a sender uses its local tail and the current value of the remote head contained in its local scratchpad to determine if a queue pair has space for an eager protocol message transfer.

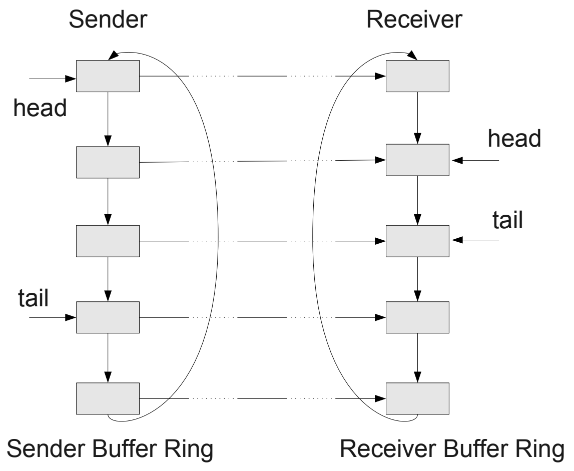

Figure 4.1: Persistent Buffer Association

The size of each buffer in a send or receive queue is a compile time constant, while the control information per peer has a size of 4 bytes (4 bytes for the remote queue head pointer value). The structure of an Eager buffer is shown in Figure 4.3.

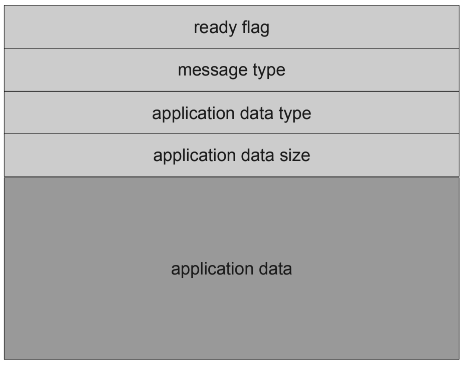

Figure 4.2: Eager buffer structure

Each eager buffer starts with a 4-byte *ready flag* used by a receiver to sense incoming data within the eager buffer at the head position of a receive queue. A 4-byte *message type* field follows and is used to differentiate among messages that can be possibly written within an eager buffer. Such messages include Eager Data and Rendezvous Control Messages. The *application data type* is a 4-byte field used to communicate the type of data that the sending MPI process transfers to the receiving MPI process. The *application data size* is a 4-byte field used to determine the amount of data to be written at the receiving process application buffers. The application data size of the sender and receiver might differ. In any case, the smallest of the two sizes is written into the receiving process application buffers.

Generally, we cannot assume that the bytes inside an eager buffer are RDMA written by the hardware in order. Such hardware features typically complicate messaging layer design. However, our platform offers counters with associated notifications. When a sender initiates an RDMA Write to a remote eager buffer, the eager buffer local to the sender is marked as being not ready and the hardware is instructed to provide a notification on RDMA completion at the remote receive buffer address, which is also the address of the ready flag. So, the sender software does not mark an eager buffer as being ready to be read by the receiver. This task is handled by the hardware and is shown in Figure 4.4. Receiver software marks again an eager receive buffer as being not ready when data within an eager receive buffer have been consumed.

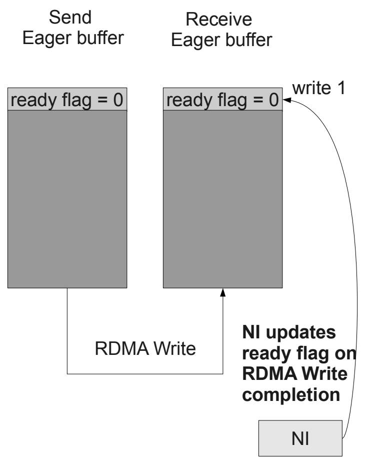

Figure 4.3: RDMA Write completion designation by NI

4.2.2 Process Synchronization Mechanism

Our MPI library requires a process synchronization mechanism to be available in order to ensure correctness of library initialization. After initialization is finished, each process is eligible to send application data to remote receivers and will initiate RDMA Write operations or Remote Stores for that purpose. So, the remote addresses must have already been allocated by the remote processes in their corresponding scratchpads or else the initialization will be erroneous.

A sense-reversing barrier implementation already existed from previous work on software development for our platform. We use this existing mechanism to synchronize processes before MPI_Init() returns to each of the process in our system. We also use the same mechanism to implement MPI_Barrier() for the communicator MPI_COMM_WORLD, which is the only communicator supported by our library.

Our work focused more on the point-to-point blocking communication primitives and less on efficient process synchronization mechanisms. For the purposes of our work, in an environment that includes only 4 processors, the existing barrier implementation was adequate to fundamentally satisfy the synchronization requirements of our library. Therefore, our MPI_Barrier() implementation does provide barrier functionality, but does not claim to provide any particular barrier performance.

It should be noted that the existing barrier implementation is based on the Xilinx Mutex core for the On-Chip Peripheral Bus (OPB). Each mutex has its own address range of 4 words of 32-bit data. The first word of the mutex can be used to implement locks and the second word for user defined data. The existing barrier implementation exploits the first word for locking and the user register for barrier data. The locking procedures that must be followed for the mutex to be locked and unlocked are described in the Xilinx manual for the OPB Mutex core.

4.2.3 Library Initialization

Library initialization is performed when an MPI application calls the MPI_Init() function. During library initialization each process performs the following tasks:

1. Assigns to the value of its own rank the value of its own CPU_ID for later support for the MPI_Comm_rank() function.
2. Initializes the number of processes in the communicator MPI_COMM_WORLD for later support for the MPI_Comm_size() function.
3. Allocates memory space in scratchpad to be used by Eager Protocol.
4. Calculates the base scratchpad memory address of remote processes.
5. Allocates an NI command buffer and space in scratchpad for the polling address of the local sender.
6. Calculates the addresses of remote eager send and receive queues and their buffers. The result is stored in cached private process variables. The caching of those values improves communication performance since those calculations are removed from the fast path.
7. Allocates space in scratchpad for tracking the head status of remote receive queues.
8. Allocates memory space in scratchpad to be used for Rendezvous Protocol and initializes the custom memory allocator that overrides malloc().
9. Enters a barrier and waits for all other processes to reach the same point before MPI_Init() returns to the MPI application calling environment.

At each step of the initialization any variables that need specific initial values are set to those values (e.g. queue head and tail pointers, buffer ready flags, etc).

4.2.4 Eager Protocol Implementation

When a sender calls the MPI_Send() function and the application message size does not exceed the Eager-to-Rendezvous threshold, then Eager Protocol will be used between the sender and receiver for the specific message transfer. It should be noted that sender decides the protocol to be used. Generally, the application message size specified in an MPI_Send() function call might be different from the application message size specified in the matching MPI_Recv() function call. In any case, the minimum of the two sizes is used to write data into the application destination buffer. Still, the sender and not the receiver decides the protocol to be used, since the sender is the only entity that knows the actual amount of data to be transferred over the network.

If sender decides that Eager Protocol should be used, then sender prepares the data for transmission by executing the following series of steps:

1. Finds available local buffer in the send queue associated with the remote receiver.
2. Fills the header of the buffer with the application message type and size. The message type is set to designate an Eager Protocol transfer and the ready flag is set to designate that the

eager buffer is not ready to be consumed by the receiver. The application data are copied to the data field within the eager send buffer.
3. The address of the corresponding remote receive queue buffer is obtained.
4. The local counter object is updated with the size of the upcoming RDMA Write, the local notification address and the remote notification address.
5. The NI is instructed to perform the RDMA Write and supply local and remote notifications.
6. The send queue local tail advances to the next posistion within the send queue ring.
7. The queue full condition is tested to determine if next Eager transfer towards the same destination rank can start immediately. Performing this test here instead of at an earlier point in this series of steps improves latency of a zero-byte Eager transfer when measured with a ping-pong test.

The receiver on the other hand does not know the protocol to be used for the transfer as described earlier in this section. The receiver does know the rank of the sending process. The receiver therefore performs the following series of steps:

1. Determines the address of the local buffer at the head of the receive queue associated with the remote sender.
2. Advances the value of the head in the local receive queue. Performing this increment here instead of at a later stage in this series of steps improves latency of a zero-byte Eager transfer when measured with a ping-pong test. Note that the sender is not yet informed about this increment, so there is no danger associated with misleading the sender with a head value that is not yet valid.
3. Polls the ready flag of the local buffer at the true head of the receive queue. This flag will be set by the NI on RDMA Write completion for the specific buffer.
4. When the local receive buffer becomes ready, receiver reads the value of the message type field within the buffer.
5. If the message type indicates that an Eager transfer is being performed, receiver copies the application data from the eager receive buffer into the application destination buffer.
6. Marks the local buffer at the true head of the receive queue as not ready.
7. Informs the sender about the new head value by performing a Remote Store into a specific location of the sender's scratchpad. Note that this update occurs for every eager transfer performed. We have also tried to employ a mechanism that would send an update only if the number of unacknowledged eager messages received has reached a value equal to half of the buffers in the send/receive queue pair (e.g. one update every 4 eager transfers if there are 8 buffers in the receive/queue pair). However, the cost of deciding whether to send an update or not is greater than performing the Remote Store itself, so we decided to simplify things and always send an update about the new head value to the remote side.

A receiver must peform a matching between a send operation with its corresponding receive operation using the source, tag and comm as has been descibed in the previous chapter. Our implementation does not support tags and MPI_COMM_WORLD is the only communicator supported. This simplifies the matching logic to a match on only the source rank. The receiver tries to detect an incoming message by polling the ready flag at the head of the receive queue dedicated to messages from the specific source rank.

Another requirement from the MPI specification is that messages should be non-overtaking. This requirement is satisfied in our implementation because of the way the send/receive queue pairs work. Each sender writes in successive locations in those queues and the receiver reads from the receive queue in order. Therefore, the application messages between a sender and a receiver are received in the same order that they were sent regardless of whether the network reorders packets or not.

It should be noted that our initial implementation used function calls for many of the steps of the Eager Protocol. However, the use of function calls in our environment can sometimes lead to high performance penalties, so we have removed most of them from the code, with only few exceptions.

4.2.5 Rendezvous Protocol Implementation

The steps of the Rendezvous Protocol have been described in the previous chapter. We provide an additional Figure 4.5 in this section for convenience while discussing the Rendezvous Protocol implementation.

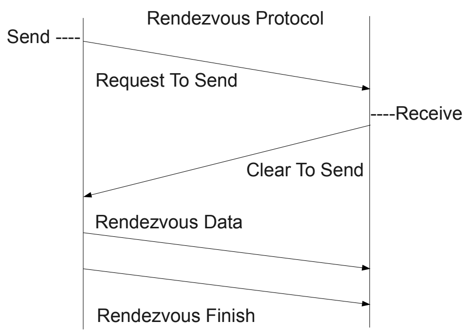

Figure 4.4: Rendezvous Protocol

The Rendezvous Protocol is used to achieve a zero-copy effect. Consequently, the application buffers at the sender and receiver sides must have been previously allocated in scratchpad regions. We have overriden malloc() with a custom mpi_malloc(). The main source code of the allocator was developed by Programming APTS and is freely available on the Internet. The logic of the allocator is similar to the well-known allocator described in [8]. The algorithm performs a first fit during allocation. We have only modified the source code to allow the allocator to be initialized and activated in the specific environment of our platform.

When we discussed the Eager Protocol implementation, we noted that only the sender knows the actual size of the messsage to be transferred over the network. When a specific application message size is crossed, sender initiates a Rendezvous Protocol transfer by sending an RTS message to the remote side using the existing Eager Protocol mechanisms. The sender then goes into a state of waiting for a CTS from the receiver (blocking send operation).

The receiver does not know the protocol to be used, but is waiting for the eager buffer at the head of the receive queue dedicated to the particular sender to become ready. When this buffer becomes ready, receiver reads the message type field within the buffer and realizes that Rendezvous Protocol has been chosen by the sender. Receiver then sends a CTS message back to the sender. CTS message contains the address of the application buffer of the receiver as well as the receiver message size and is sent as an Eager Protocol message. The application buffer has been previously allocated in scratchpad memory using a custom malloc() implementation and such a buffer is capable of being RDMA written directly by the sender without copies being necessary. After the receiver sends the CTS, it goes into a state of waiting for the FIN from the sender (blocking receive operation).

When sender receives the CTS, it extracts the remote address and receiver message size from the eager buffer and performs an RDMA Write directly to the receiving application's buffer. When the RDMA operation is complete, receiver could be notified by an independent eager FIN message from the sender that would signal the end of the Rendezvous Data transfer. In order to avoid the overhead of an additional eager message for the FIN, we exploited our hardware platform notification mechanism. When the sender prepares to instruct the NI to send the Rendezvous Data, it also sets the receiver notification address to be the address of the next eager receive buffer the receiver will consume from the particular sender. Therefore, the hardware automatically updates the receiver on completion of the data transfer and the sender does not send an additional control message for the FIN. The receiver detects completion by checking the head of the receive queue for the particular sender and the Rendezvous is finished.

4.3 Library Implementation Summary

4.3.1 Scratchpad Partitioning

Each L2 cache is 4-way set-associative, with each of the ways being 16Kbytes, which yields a total of 64Kbyte L2 cache size. We cannot use all the ways of the cache as scratchpad or a program which uses cached variables will not be able to function. In addition, the more cache ways we allocate as scratchpad, the less the performance of the cache will be, since its available size will be reduced. We therefore have chosen to allocate a single way of the cache to be used as scratchpad for the purposes of the Eager and Rendezvous Protocols.

Each eager buffer has a size EAGER_BUFFER_SIZE and each send or receive queue contains EAGER_BUFFER_QUEUE_LENGTH such buffers. Each processor allocates 1 send queue per remote receiver and one receive queue per remote sender. Each processor does not allocate queues for sending and receiving eager messages from themselves. The total scratchpad memory size allocated for the system buffers required for the operation of the Eager Protocol is therefore: 2*EAGER_BUFFER_SIZE*EAGER_BUFFER_QUEUE_LENGTH*(processors–1). For an eager buffer size of 64bytes and an eager queue length of 8 entries, the scratchpad space allocated for the eager queues at each processor is : 2 * 64 * 8 * (4-1) = 3072 bytes. The number of eager buffers for each queue and the size of the eager buffers can be configured at compile time.

Because each processor does not allocate eager buffers for themselves, each processor must map each processor rank to the base addresses of the remote eager buffer receive queues and the base addresses of the local send queues for sending to a remote rank. This task is performed during library initialization.

We also allocate 1 Counter (32 bytes), 1 NI Command buffer (32 bytes), 1 sender polling address (32 bytes) and per remote receiver space for the head values of the receivers to be written back to the sender (1 cacheline or 32 bytes). The total space for these additional allocations is 128 bytes.

After all the previous requirements have been satified, the remaining space of the 16Kbyte scratchpad is left to initialize the memory allocator used for Rendezvous. The allocator requires a base address and a total available memory size for initialization.

4.3.2 Current Limitations

We do not provide a full MPI standard implementation. Our libary supports only the following MPI functions: MPI_Init(), MPI_Comm_size(), MPI_Comm_rank(), MPI_Send(), MPI_Recv(), MPI_Barrier() and MPI_Finalize(). We do not support MPI_ANY_SOURCE, the use of tags, derived datatypes, and the status object in MPI_Recv() arguments. MPI_COMM_WORLD is the only communicator supported. The MPI_Send() implementation is blocking until the data have been delivered to the remote scratchpad and a notification is provided by the hardware to the sender. Note that in the case of Eager Protocol this does not necessarily mean that the matching MPI_Recv() has copied the data from an eager buffer to the application buffer of the receiver. In the case of Rendezvous, and due to the zero-copy procedure, when the sender is notified about completion of the network data transfer it also means that the data have actually been delivered to the receiver application buffer. Therefore, when the Rendezvous Protocol is chosen by the sender, the semantics of MPI_Send() become those of a synchronous send.

4.3.3 Software Architectural Constants

The number of eager buffers in each send and corresponding receive queue is a compile-time configurable parameter of our library. The importance of this parameter can be explored in an implementation that does not block the sender until the data have been copied by the hardware to the remote scratchpad. Our implementation blocks the sender until a notification is delivered by the NI at an agreed address, so using many eager buffers at each queue pair does not make a difference to the observed performance.

Theoretically, we could block the sender until the NI has written a zero value to the address of the NI command buffer. However, at the time of development of our library this feature of the hardware was not supported. Besides, even if the NI command buffer can be reused early, the MPI library also needs to know when the Counter at the sender can be reused. Therefore, future implementations could support a pool of Counters without necessarily supporting a pool of NI command buffers, since the NI command buffers are expected to be consumed faster than the notifications are delivered.

The size of an eager buffer is also a compile-time configurable parameter of our library and and the same holds for the threshold between the Eager and Rendezvous Protocols. Those two parameters are closely related. A typical design choice would be for the threshold to be set to the size of the payload that can be accommodated by an eager buffer. The size of this payload still needs to be tuned and we will revisit this issue after we describe the latency results obtained from our performance measurements for the Eager and Rendezvous Protocols.

4.4 Performance Measurements

4.4.1 Latency of Eager Protocol

We measure the 1-way latency of the Eager Protocol by performing an experiment similar to the experiment used in [16]. We have one CPU of our system execute MPI_Send() followed by MPI_Recv() while a second CPU excutes the matching MPI_Recv() and MPI_Send() in a ping-pong fashion. The ping-pong test is repeated 10,000 times (with 10 warmup repetitions). The 1-way latency is computed as the total time divided by 20,000, which corresponds to half the average round-trip time (½ RTT). Our measurements were obtained in units of clock cycles.

We have set the eager buffer size to 512 bytes and executed the ping-pong experiment for application message sizes within the interval from 0 to 496 bytes in 2 byte increments . The header of an eager buffer is 16 bytes, so the actual data transferred by the network are within the interval from 16 to 512 bytes. Some sample values of the 1-way latency are provided in Table 4.1.

Application Message Size (bytes)	**1-way Latency (clock cycles)**
0	217
16	644
48	737
112	924
240	1297
496	2065

Table 4.1: Eager Protocol 1-way latency

The results of our experiment are plotted in Figure 4.5.

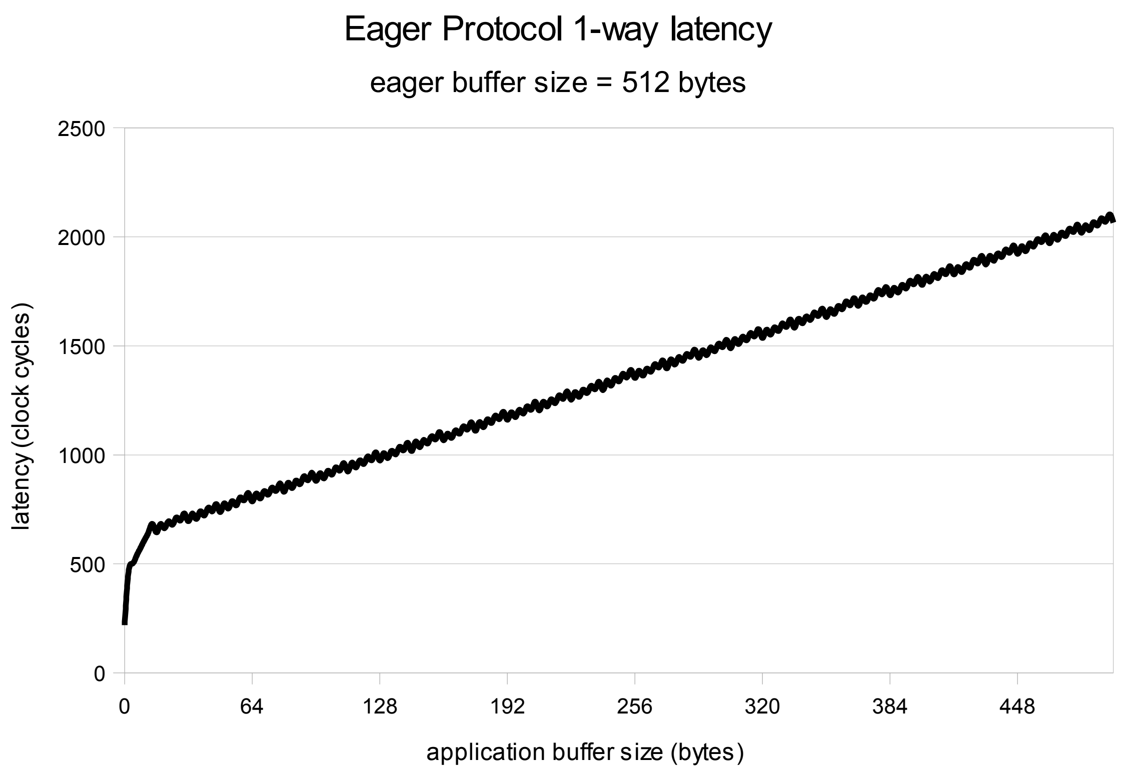

Figure 4.5: Eager Protocol 1-way latency

The zero-byte latency refers to an application message size of 0 bytes and a network message of 16 bytes. When a zero-byte application message is transferred, the Eager Protocol does not perform a copy from application to eager buffers and vice versa. Therefore, the zero-byte latency represents a base latency for all application messages transferred. The base latency includes software overheads and the overhead of the hardware for sending a 16-byte message and providing notifications.

All application messages with size greater than zero have a latency greater than the base zero-byte cost, while the latency increases as application message size increases. The variable part of the latency is mainly due to one copy at the source from the application send buffer to the eager send buffer and a second copy at the destination from the eager receive buffer to the application receive buffer. We perform copies at the sender and receiver using memcpy(). The sender performs a series of stores to scratchpad from cached data. The receiver performs a series of stores from scratchpad to cached data. Accesses to DRAM are also possible while the previous procedures are being performed.

Another reason for the increase in the observed latency with message size is the fact that the hardware delay for RDMA Write operations between scratchpads also increases with message size, although this factor is not the dominating reason for the linearly and quickly increasing latency observed in Figure 4.5. The analysis of Rendezvous in the following paragraphs will show that the hardware communication delay does not increase so fast with message size.

The zero-byte 1-way latency of our implementation is 217 clock cycles. The Cell Messaging Layer has a zero-byte latency of 870 SPE clocks within the Same Cell according to [17]. It must be noted that the two platforms are different both from processor and network capabilities perspective.

The tremor in Figure 4.5 is related to how the hardware handles a number of bytes that are not multiples of 4 bytes. In Figure 4.6 we plot again the values for smaller application message sizes and it is clear that non-multiples of 4-bytes experience higher delay than the next larger multiple of 4 bytes.

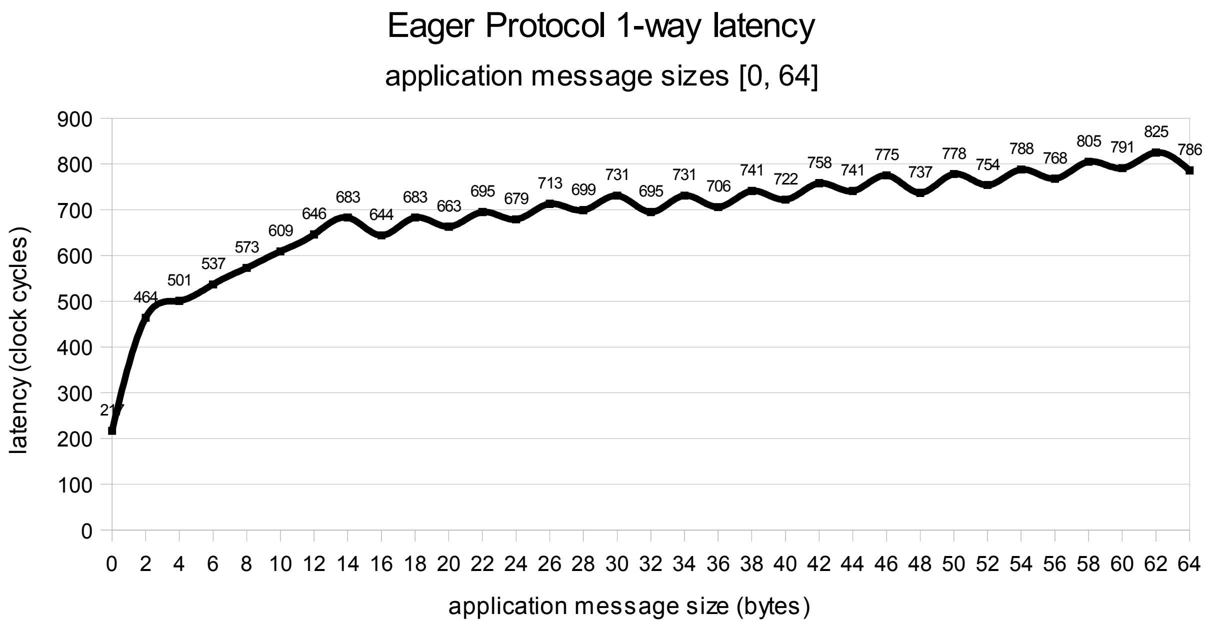

Figure 4.6: Eager Protocol latency for small message sizes

Figure 4.6 will also be useful when we decide where to set the threshold between Eager and Rendezvous Protocols.

4.4.2 Analysis of Eager Protocol RTT Latency

In the previous section we provided an estimation of the Eager Protocol 1-way latency. The 1-way latency was computed as an average and hides the latency of both the send and the receive path. In general, it is expected that the latency of the send and the receive path will not be equal, so we executed again the ping-pong experiment to measure the latency components of the zero-byte Eager transfer during the ping-pong test. We sampled various checkpoints within the MPI_Send() and MPI_Recv() source code and recorded the checkpoints of the last iteration of the ping-pong loop, which are shown in Figure 4.7.

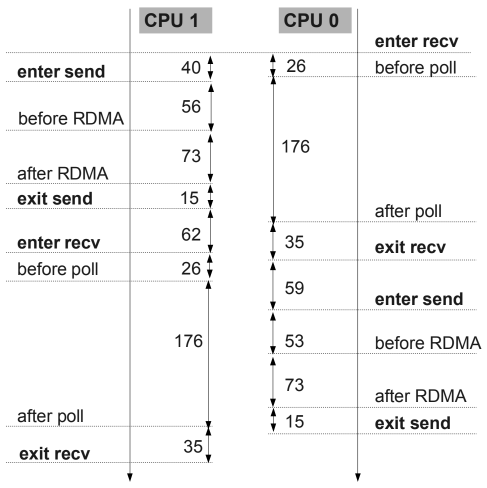

Figure 4.7: Analysis of Eager Protocol latency

The cycle counts in Figure 4.7 are indicative. Each run of the ping-pong test will yield slightly different results. The number of calls to timers affects the results and access to cached variables inserts a degree of non-determinism to the cycle counts reported. However, the results are useful in order to obtain an estimation of the latency components of the Eager Protocol and to determine the contribution of each of those components to the total delay observed when running the ping-pong test.

CPU 1 on the left of Figure 4.7 spends 144 cycles on MPI_Send(), 237 cycles on MPI_Recv() and 62 cycles between MPI_Send() and MPI_Recv(). The sum of those delays is 443 cycles. CPU 0 on the right of Figure 4.7 spends 237 cycles on MPI_Recv(), 141 cycles on MPI_Send() and 59 cycles between MPI_Send() and MPI_Recv(). The sum of those delays is 437 cycles. The zero-byte 1-way delay reported with the ping-pong test is 217. The average of 443 and 437 is 440 and divided by two gives us 220 cycles, which is close to 217 (220 includes timer overheads).

The process running on CPU 0 enters MPI_Recv() 40 cycles before the process running on CPU 1 enters MPI_Send(). The process on CPU 0 starts polling at the local notification address in scratchpad 26 cycles later. CPU 0 spends 176 cycles out of the total 237 cycles of MPI_Recv() on polling. The 176 cycles of polling correspond to slightly above 74% of the total MPI_Recv()

latency. Besides polling, CPU 0 spends 61 cycles for other work, such as buffer management and flow control. More details about the software overheads of MPI_Recv() will be provided in the sections that follow.

The process running on CPU 1 spends 73 out of the total 144 cycles of MPI_Send() on issuing the 4 stores required to start an RDMA Write and polling on the local notification address. The 73 cycles correspond to slightly above 50% of the MPI_Send() total time. The remaining 71 cycles include Eager Protocol buffer management, RDMA operation preparation and flow control. More details about the software overheads of MPI_Send() will be provided in the sections that follow.

As we have seen, both the sender and the receiver spend time on polling. Our current implementation includes only blocking functions and we have implemented them as such. Each sender allocates only 1 NI command buffer and only 1 Counter to be used for all possible receivers. The polling time at the sender might be improved with the use of a pool of RDMA command buffers and a pool of Counters. Such pools will require additional memory in scratchpad however. So, future work could trade memory for speed to the extent possible in our platform. Still, those pools will have to be managed by software and additional software overheads might be introduced.

Assuming that t=0 corresponds to the moment CPU 0 enters MPI_Recv(), then CPU 1 reaches the point "before RDMA" at t = 40 + 56 = 96 and the point "after RDMA" at t = 96 + 73 = 169. CPU 0 reaches the point "after poll" at t = 176 + 26 = 202. So, it takes 202-169 = 33 more clock cycles for CPU 0 to detect that the RDMA is finished than it takes for CPU 1. Part of those cycles include the overhead of issuing the 4 stores on CPU 1. Assuming that those are the theoretical 4 clock cycles, then it takes 29 more cycles for CPU 0 to detect the RDMA completion. This duration for CPU 1 is 73-4 = 69 clock cycles (16-byte RDMA takes 34 clock cycles according to [5] and we also wait for notification from a local Counter).

In our current implementation, when an RDMA Write is complete, the hardware updates the Counter at the sender and a notification is triggered back to the receiver. The polling time at the receiver might be reduced with Counters being allocated at each receiver instead of at each sender. The logic of this suggestion is related to the way the hardware provides the notification mechanism. If Counter is allocated at the receiver, then hardware can provide a notification to the receiver as soon as data have been written to its scratchpad. However, such a design will require per-sender Counters at each receiver (which implies additional memory and further scalability concerns for the Eager Protocol) or a locking mechanism for the Counter of the receiver.

A process that enters MPI_Recv() before the sender is ready will have to poll at least until the sender enters MPI_Send() and we cannot do anything to improve this situation besides providing non-blocking operations in the future. In the general case of an application that also includes computation, the polling time might or might not be avoided, even when using non-blocking operations (e.g. when the expected data are absolutely required for the application to proceed any further). In this specific ping-pong test however, the polling time affects the measurement of the RTT and consequently the measurement of the 1-way latency. As it can be seen from Figure 4.7, each process spends 141-144 cycles to send data, 237 cycles to receive data and there exists an additional overhead of 59-62 cycles between the MPI_Send() and MPI_Recv() calls. This overhead must be function call and return overhead because there is no other code between exit from MPI_Send() and entrance of MPI_Recv() and vice versa. As we have seen in other cases while developing the code for the MPI library, the cost of function calls in our FPGA environment can sometimes be high. The program .text is loaded from DRAM. We are using an instruction cache, but misses might still occur. More importantly, the program stack is also loaded from DRAM via the data cache path, so misses can occur there as well with the result being the DRAM latency penalty.

4.4.3 Latency Components of Eager Protocol

Table 4.2 shows the latency in clock cycles of components within MPI_Send() when using the Eager Protocol to send zero-byte application messages (16-bytes RDMA for the Eager Protocol header) with a ping-pong test performing 10,010 iterations. We are sampling only within MPI_Send(), computing differences among checkpoints. As we have seen in general, the more points we sample, the more intrusive the testing becomes and affects all measurements, since the measurement overhead in itself becomes significant and there exist synchronization dependencies between the processors during the ping-pong test that affect processor waiting times.

The sampling for the Eager Protocol within MPI_Recv() is shown in Table 4.3, and the results are enlarged by the intrusive measurement. The values in Table 4.3 have been measured while running the same ping-pong test we used for MPI_Send(), but we are now sampling only the MPI_Recv() function at many points, computing the differences of adjacent points and summing them up for the duration of the loop. The numbers include some computation/storing overhead for the sum of differences between sample points and are tracked across many calls to MPI_Recv().

We have conducted other tests where we sampled less points within MPI_Send() and MPI_Recv() with 10010 iterations and the results of the averages are shown in Table 4.2 and Table 4.3 in bold. The other numbers can be used as an approximation of each operation's weight in the total cost. Note also that the numbers in the timing sequence for the Eager Protocol that we described in the previous section were also different, since they represented a single iteration of the ping-pong loop and we were not storing values from iteration to iteration (less intrusive test, but maybe not representative).

Software Operations for Eager Send	**# clock cycles**
1. Calculate message size in bytes from count (branch) 2. Choose between Eager and Rendezvous Protocols (branch) 3. Read tail of sendq (cached memory accesses) 4. Increment tail (arithmetic operation) 5. Read address of remote head (cached memory access) 6. Read address of send buffer (cached memory access 7. Fill eager buffer header (scratchpad memory accesses) 8. Check if message must be copied (branch)	40
1. Determine remote eager buffer address (cached memory access and arithmetic operation) 2. Calculate RDMA size (arithmetic operation) 3. Read Counter address (cached memory access) 4. Store RDMA size to local Counter (scratchpad write) 5. Store notification addresses to local Counter (scratchpad write)	22
Before RDMA Write	62 **(53)**
1. Issue 16-byte RDMA Write (4 stores) 2. Poll for completion	78 **(77)**
1. Update tail (cached memory access, branch) 2. Check queue full condition (scratchpad memory access, branch)	25 **(17)**
After RDMA Write	25 **(17)**

Table 4.2: Eager Send Latency Components

Software Operations for Eager Receive	**# clock cycles**
Calculate message size in bytes from count (branch)	7
Read address of receive buffer (cached memory accesses)	18
Store address of ready flag to volatile variable	7
Increment head of receive queue (cached memory access, branch)	11
Before polling	43 **(33)**
Poll on volatile variable (branch)	Varies (sample value 166)
Choose between Eager and Rendezvous Protocols (branch)	27
Decide to copy user buffer data to eager buffer payload (branch)	9
Invalidate receive buffer (scratchpad write)	8
Update remote head with our new head value (cached memory accesses, Remote Store)	15
After arrival of data	59 (**45**)

Table 4.3: Eager Receive Latency Components

4.4.4 Latency of Rendezvous Protocol

We have run the ping-pong test to measure the average latency of the Rendezvous Protocol. The Eager to Rendezvous threshold was set to 32 bytes. The results are presented in Figure 4.8.

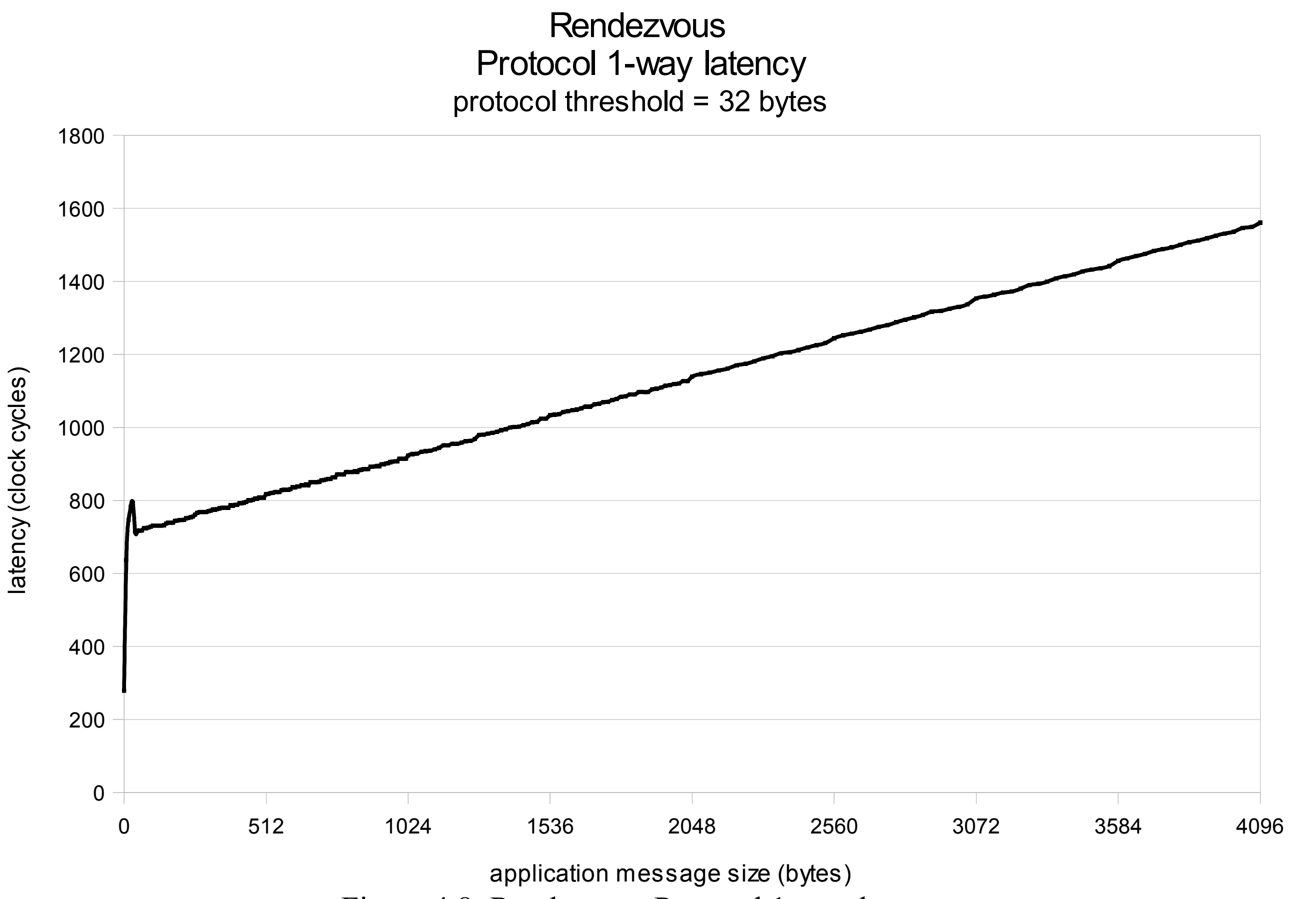

Figure 4.8: Rendezvous Protocol 1-way latency

When the Eager to Rendezvous threshold is set to 32 bytes, the Rendezvous is used for application messages larger than 32 bytes, while the Eager Protocol is used for application message sizes smaller or equal to 32 bytes. We have used malloc() to obtain a 4096 byte application buffer in scratchpad and we use the same buffer to send application messages from 0 to 4096 bytes. A spike is observed for small message sizes. This spike for message sizes up to 32 bytes does not represent the Eager Protocol latency. Because the application buffer has been allocated in scratchpad, the Eager Protocol performs copies from application buffer in scratchpad to eager buffer also in scratchpad and vice versa instead of between cached application buffer and scratchpad eager buffer. For the purposes of this test we can ignore the Eager Protocol measurements, since we are trying to analyse the performance of the Rendezvous Protocol in this experiment.

Application Message Size (bytes)	1-way Latency (clock cycles)
48	716
64	718
128	731
256	761
512	817
1024	923
2048	1139
4096	1561

Table 4.4: Rendezvous Protocol 1-way latency

From Figure 4.8 it becomes obvious that the Rendezvous Protocol outperforms the Eager Protocol from very small application message sizes. Some sample latency values are reported in Table 4.4. The Eager Protocol has a latency of 2065 cycles when the application buffer size 496 bytes, while the Rendezvous transfers 512 bytes in 817 clock cycles. The Rendezvous can transfer 4096 bytes in less clock cycles (1561) than the Eager can transfer 496 bytes (2065). It is also clear that the hardware overhead when message size increases does not grow as fast as the Eager Protocol copy overhead grows.

We plot the Rendezvous Protocol latency for small message sizes in Figure 4.9. If we compare this figure to Figure 4.4, we see that, for message sizes between 32 and 48 bytes bytes the two protocols compete, but for sizes greater than 48 bytes the Rendezvous Protocol always performs better from a latency perspective.

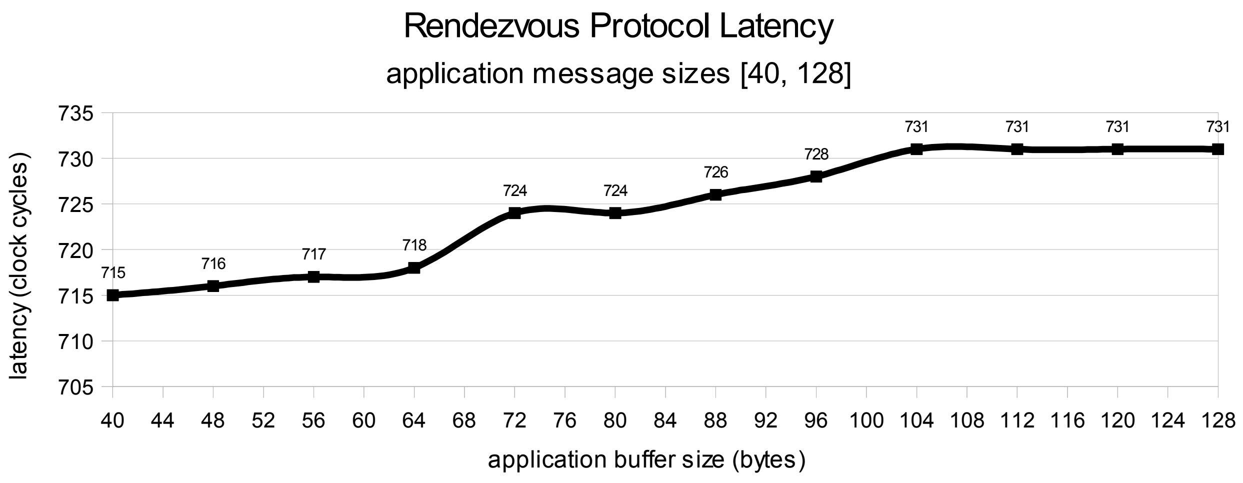

Figure 4.9: Rendezvous Protocol latency for small message sizes

4.4.5 Analysis of Rendezvous Protocol RTT Latency

In Figure 4.10 we show averages of latency of the Rendezvous Protocol components during the ping-pong test. We have set the Eager to Rendezvous threshold to 48 bytes and used an application message size of 64 bytes for the actual Rendezvous Data transfer. The Figure 4.10 is not to be strictly interpreted as a timing sequence ordering the stages of the protocol in the two processors. It is more meant to describe the difficulties in estimating messaging layer overheads in terms of execution time. One of the reasons the authors of [6] have used instruction counts in their analysis is of messaging layer costs is the difficulty in obtaining accurate cycle counts due to the details of the hardware. We have run the ping-pong test many times and the cycle counts we obtain vary significantly. If we obtain single instances of the ping-pong loop and try to analyse a timing sequence, it is nearly impossible to argue using the only the numbers reported about why each component of the instance lasts a particular number of cycles and why in another instance it lasts a different number of clock cycles. It is typically difficult to argue about performance in a system that contains caches even when a single processor is used. With more processors, the issue of non-determinism becomes more evident.

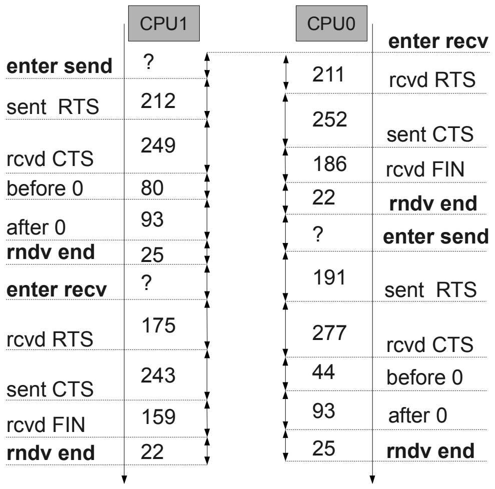

Figure 4.10: Analysis of Rendezvous Protocol latency

We therefore have computed averages only. We obtain the differences among sample points within MPI_Send() and MPI_Recv(), we sum the differences and when the ping-pong loop is done with the iterations we compute the average of the differences that we summed up during the loop. All the numbers in Figure 4.10 are average latencies of the components.

Since the results vary and we have computed only averages, there are two difficulties in analysing the timing sequence of the execution of the program in the two processors. One difficulty is to decide which processor enters the sequence first. Another difficulty is related to function call overhead. We have performed another ping-pong test that measures the average duration of MPI_Send() and MPI_Recv() in each of the processors. During this second test we have removed all the timing code from within MPI_Send() and MPI_Recv(). We only use timers within the ping-pong loop that invokes MPI_Send() and MPI_Recv().

The average MPI_Send() duration on CPU 1 is 735 clock cycles, while the sum of the average components shown in Figure 4.10 is 659 clock cycles. We therefore have a difference of 76 clock cycles. We have confirmed from individual instances of the loop that such differences always exist from the exit of MPI_Send() until we enter MPI_Recv() code and vice versa. In addition, the average MPI_Send() duration on CPU 0 is 687 clock cycles, while the sum of the average components is 599, which yields a difference of 57clock cycles.

The average MPI_Recv() duration on CPU 1 is 664 clock cycles, while the sum of the average components shown in Figure 4.10 is 599 clock cycles. We therefore have a difference of 65 clock cycles. In addition, the average MPI_Recv() duration on CPU 0 is 689 clock cycles, while the sum of the average components is 671, which yields a difference of 18 clock cycles.

If we sum all the average components on CPU 1, the total is 1258 clock cycles. If we further add the MPI_Send() and MPI_Recv() average function call overheads for CPU 1, then the total time in an average iteration for CPU 1 is 1399 clock cycles. If we follow a similar procedure on CPU 0, then the total average time becomes 1301+57+18 = 1376 clock cycles. The largest of the two total times is 1399 and if we divide it by two, the result is 700 clock cycles, which is close to the minimum 1-way latency reported for Rendezvous, and is in fact smaller. The ping-pong loop also includes the overhead of the loop itself and a test condition for the initial warmup iterations, so after taking all this into account we are quite close to at least justify at a high level the numbers obtained for the individual components.

Despite the difficulty in analysing an exact timing sequence, we have at least a few synchronization points in the Rendezvous Protocol to help us start. We do not know which processor begins first, but it is clear that MPI_Send() on CPU 1 has sent the RTS after a duration of 212 clock cycles, while MPI_Recv() on CPU 0 has received the RTS after a duration of 211 clock cycles. This first step can be considered to have in both processors a delay approximately equal to the average 1-way delay of a zero-byte Eager transfer (the RTS is equivalent to an eager zero-byte message).

The second step in the Rendezvous Protocol is for CPU 0 to send the CTS. CPU 0 spends 252 clock cycles to send the CTS, while it takes 249 clock cycles for CPU 1 to receive the CTS. This step can be considered to have in both processors a delay approximately equal to the average 1-way delay of a 4-byte Eager transfer, because an eager CTS message includes the 4-bytes of the receiver application buffer address within the body of the eager message. In the case of the CTS message, the network will send 20-bytes of total data and not 16 bytes as is the case with a zero-byte application message, and we also pay some overhead for writing into the data portion of the eager buffer in scratchpad.

The third step is for CPU 1 to prepare to instruct the NI to perform the zero-copy. During preparation we observe a delay of 80 clock cycles, while the same software step at a later point in CPU 0 lasts 44 cycles on average and both CPUs are running the same code. This software step includes accesses to cached variables in order to prepare the Counter and NI command. The actual execution time might depend on misses on individual processors and/or the timing of the misses and whether misses occur simultaneously on the two processors. During this step CPU 0 determines where to wait for the Rendezvous FIN, performs some buffer management tasks and starts polling at its local notification address. We have moved some buffer management tasks at this stage, to reduce polling time and to make the next step faster for CPU 0 (i.e. CPU 0 will have less remaining work to do after it receives the FIN). Note also that CPU 1 does not send a separate FIN eager message, but this task is rather handled by the RDMA Write notification mechanism.

The fourth step on CPU 1 includes the 4 stores required to issue an RDMA Write and then CPU 0 starts polling at its local notification address. The whole step lasts 93 cycles and the same duration is reported at a later equivalent point in the sequence for CPU 0. If we assume that the 4 stores last 4 clock cycles, then it takes 93-4 = 89 clock cycles for an RDMA Write of 64 bytes to be performed and for the local Counter on CPU 1 to trigger a notification at an address in the scratchpad of CPU 1. According to [5], an RDMA Write of 64 bytes takes 52 clock cycles. So, we are left with 89-52 = 37 clock cycles that must be associated with the triggering of a notification. Those 37 clock cycles will include the time for CPU 1 to notice, while being in a loop testing a volatile variable, whether the RDMA Write is complete. Note also that there are 2 notification addresses configured on the Counter local to CPU 1 (one local and one remote notification address). CPU 0 on the other hand, is informed about completion 186 cycles after it started polling, but the difference between the time it was informed and the time the RDMA Write started cannot be computed, since we do not have an exact timing sequence.

The last step before CPU 1 exits MPI_Send() lasts 25 clock cycles and includes buffer management overhead. The tail of the send eager queue is incremented (cached variables are accessed, branch for buffer ring wraparound), the eager buffer that CPU 1 received the CTS is marked as not ready (scratchpad write) and the remote peer is informed about the new head of our receive queue (Remote store). Similar steps are performed in CPU 0 (FIN buffer marked as not ready via a scratchpad write and Remote Store for receive queue head).

The next step is for CPU 1 to enter MPI_Send() after paying a function call overhead, while CPU 0 enters MPI_Send () and the steps described previously are reversed. We notice that the sending of RTS is performed faster than it did when CPU 1 had sent it, while the sending of CTS is performed slower. Our code does include function calls for sending the RTS and CTS, so some variability can occur there. It was not possible to remove those function calls because the code would become hard to debug. In fact, the function calls have been put to resolve a bug that we could not have resolved otherwise. We have generally avoided function calls, but those two have not been removed for the reasons just described. It must be noted though, that the increased variability in the delays of the Rendezvous Protocol components that was described at the beginning of this section can be attributed to those function calls, as well as, to the 3 steps of the protocol, which make the analysis harder than the corresponding analysis of the Eager Protocol.

The Rendezvous Protocol can roughly be viewed as experiencing a base delay of a "virtual" RTT comprising of a combination of the 1-way delay of a zero-byte Eager message (217 for RTS) and the 1-way delay a 4-byte Eager message (501 for CTS, but we do not call memcpy() in this case). We see that both processors experience an initial delay of 461-463 clock cycles before the actual Rendezvous Data transfer begins. The zero-copy sequence we analysed was for a 64-byte application message, which has a computed average 1-way delay of 718 clock cycles. Approximately 718-462 = 256 clock cycles are left. The network overhead is in the order of 93 cycles, so we have 163 clock cycles due to: software overheads for buffer management and NI command preparation, variability in function call overheads for RTS/CTS and the MPI_Send()/MPI_Recv() themselves, idle polling time at the receivers during the zero-copy data transfer, and loop overheads for performing the ping-pong test itself.

4.4.6 Eager to Rendezvous Threshold

The Eager Protocol can be used for application messages sizes up to 32 bytes that correspond to RDMA Write operations of up to 48 bytes. From there on, the overhead of copies from application buffers to system buffers at the sender and from system buffers to application buffers at the receiver exceeds the overhead of using the Rendezvous Protocol. It must be noted that from 32 up to 48 bytes of application message size (64 bytes RDMA Write) the protocols compete.

On the other hand, the use of Rendezvous Protocol incurs the initial overhead of 2 software protocol steps before the actual application data transfer occurs. The latency of Rendezvous Protocol for application messages smaller than 32 bytes exceeds the latency of the Eager Protocol.

We could set the Eager to Rendezvous Protocol threshold to 32 bytes. At an early stage of the implementation we had set the Eager to Rendezvous threshold to be equal to the eager buffer data size. The size of each eager buffer can now be set to 64 bytes to make it equal to the size of 2 cachelines, while the Eager to Rendezvous threshold can be disassociated from the eager buffer data size and set independently at compile time to be equal to 32 bytes. Note that this implies that 16 bytes of each eager buffer will be wasted (32 bytes eager data + 16 bytes header = 48 bytes).

Another consideration regarding the Eager to Rendezvous threshold is related to the way the Rendezvous-capable application buffers are allocated. In order to achieve the zero-copy effect, an MPI application must call malloc() to obtain a buffer in scratchpad. Every time an MPI application calls malloc() it does not necessarily need a buffer in scratchpad with the goal to communicate using RDMA operations. So, the threshold could be set to 48 bytes or higher to avoid scratchpad allocations for very small buffers. At this point, it seems reasonable to set the threshold to 48 bytes since the scratchpad space for the eager buffers will be wasted anyway if we set the threshold to 32 bytes. Increasing the threshold further presents a tradeoff between memory and speed.

Chapter 5

Conclusions

In this master thesis we present an implementation of basic MPI primitives in a multicore FPGA platform. The platform includes 4 RISC-type processors interconnected via a NOC. Each of the processors incorporates an L2 data cache tightly coupled to the processor. Each L2 data cache can be configured at run time to operate partly as cache and partly as scratchpad. Each L2 data cache is integrated with an NI. The NI connects each processor to the NOC and can be controlled by software to facilitate message exchange between the processors.

We have implemented basic MPI primitives using the operations provided by the specific hardware to the software. We provide blocking MPI message exchange primitives only. We have implemented the Eager and Rendezvous Protocols, which are commonly used internally in MPI libraries to provide the send/receive functions to upper software layers. The choice between Eager and Rendezvous Protocols is decided by the sender at run-time based on the size of the application message being tranferred.

The implementation of Eager Protocol uses RDMA Write to transfer data eagerly to the remote side and Counters with associated notifications to detect data transfer completion. It also uses Remote Stores for each receiver to inform remote senders about the corresponding receive queue status.

The implementation of the Rendezvous Protocol is RDMA Write based. The Eager Protocol is used before the actual Rendezvous data transfer for sending the RTS and CTS control messages required for the sender to be informed about the remote address to write to. The actual Rendezvous data transfer uses RDMA Write to perform a zero copy from the application buffer of the sender to the application buffer of the receiver. The FIN control message is not sent independently by the software, but the same effect is accomplished by instructing the hardware to provide a notification to the receiver using Counters.

The performance measurements taken indicate that the offered hardware operations (RDMA Write, Remote Store, Counters) can help develop low latency messaging layer software over the specific platform. The zero-byte latency of our MPI library is 217 clock cycles in an FPGA prototype with a 75 MHz clock. We have used the results of the performance measurements to set the threshold between Eager and Rendezvous Protocols to 48 bytes, while the size of each eager buffer is set to 64 bytes to make it equal to the size of 2 cachelines. For application message sizes up to 48 bytes the Eager Protocol will be used, while for application message sizes above 48 bytes the Rendezvous Protocol will be used to achieve a zero copy effect.

Future work will aim to extend the current work in the following ways:

- Implement the Rendezvous Protocol using RDMA Read operation instead of RDMA Write for the actual zero-copy data transfer, with the goal to reduce the software protocol steps and increase overlap of communication with computation in order to improve MPI application performance further.
- Implement non-blocking MPI primitives. The challenge will be to keep any additional required data structures to a minimum to facilitate simplicity and low latency in the messaging layer, while also taking into account the limited size of scratchpad regions and the DRAM latency.

Bibliography

[1] J. Duato and S. Yalamanchili, L. NI. *Interconnection networks: an engineering approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[2] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, et al. Introduction to the Cell multiprocessor. In *IBM Journal of Research and Development*, July-September 2005.

[4] InfiniBand Trade Association. http://www.infinibandta.org/

[5] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang. FPGA implementation of a configurable cache/ scratchpad memory with virtualized user-level RDMA capability. In *International Symposi*um on Systems, Architectures, Modeling, and Simulation, 2009.

[6] V. Karamcheti and A. A. Chien. Software overhead in messaging layers: where does the time go? In *Proceedings of ASPLOS-VI,* San Jose, California, October 5-7, 1994.

[7] M. Katevenis. Interprocessor communication seen as load-store instruction generalization. In *The Future of Computing, essays in the memory of Stamatis Vassiliadis, K. Bertels e.a. (Eds.), Delft, The Netherlands,* pages 55-68, Sept. 2007.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (2nd ed.).* Prentice Hall Inc., 1988.

[9] J. Liu, J. Wu, S. P. Kini, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. K. Panda. MPI over InfiniBand: Early Experiences. *Technical Report, OSU-CISRC-10/02-TR25.,* January, 2003.

[10] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS 04)*, April, 2004.

[11] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing*. San Francisco, Bay Area, June, 2003.

[12] Mellanox Technologies. http://www.mellanox.com

[13] Message Passing Interface Forum. http://www.mpi-forum.org/index.html

[14] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[15] MVAPICH: MPI over InfiniBand. http://mvapich.cse.ohio-state.edu

[16] G. Nikiforos. *FPGA implementation of a cache controller with configurable scratchpad space.* Master's thesis, University of Crete, Department of Computer Science, Heraklion, Greece, January 2009.

[17] S. Pakin. Receiver-initiated message passing over RDMA networks. In *Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, Florida, April 2008.

[18] M. Saldana and P. Chow. TMD-MPI: an MPI implementation for multiple processors across multiple FPGAs. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL 2006)*, pages 329-334, August 2006.

[19] C. Sapuntzakis, A. Romanow, and J. Chase. The Case for RDMA. *Internet Draft*. http://tools.ietf.org/id/draft-csapuntz-caserdma-00.txt, December 2000.

[20] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, New York, 2006.

[21] The Cell project at IBM Research. http://www.research.ibm.com/cell/

[22] G. Varghese. *Network algorithmics: an interdisciplinary approach to designing fast networked devices.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.