TECHNICAL UNIVERSITY OF CRETE

Hardware Accelerated Basic Blocks for Power-Aware Intercommunication in HPC and Embedded Systems

Nickos Tampouratzis

e-mail: ntampouratzis@isc.tuc.gr

Supervisor Professor: Papaefstathiou Ioannis

Professor Pnevmatikatos Dionisios

Professor Dollas Apostolos

Co-Supervisor: Dr. Mattheakis Pavlos

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

AT

TECHNICAL UNIVERSITY OF CRETE

CHANIA, GREECE

MAY 2014

Abstract

In the past, a transition to the next fabrication process typically translated to more transistors and frequency and less power. The higher frequencies paired with innovations in computer architecture defined the semiconductor industry and research until the mid-90s. At that point architecture research saturated and industry resided to the technology scaling for performance gains. During the mid-00s frequency scaling saturated as well. Transistor count, the only resource which reliably kept scaling, along with intra-chip parallelism, which could leverage and extend the existing knowledge of old-days supercomputers, emerged as the only solution to keep Moore's law live. In parallel systems, computing nodes cooperate to solve processing intensive problems. The communication between nodes is achieved through a variety of protocols. Traditionally, research has focused on optimizing these protocols and identifying the most suitable ones per system and application. Recently, an attempt to unify the primitive operations of the proposed intercommunication protocols has been realized through the Portals system. Portals offer a set of low level communication routines which can be composed to model complex protocols. However, Portals modularity comes at a performance cost, as communication protocols have been tuned and many of their timing critical parts have been decoupled from the main execution thread and in many cases accelerated as dedicated hardware. This work targets to close the performance gap between a generic and reusable intercommunication layer, Portals, and the several monolithic but highly tuned protocols. A software driven hardware accelerated system is suggested which resides on execution of actual software to highlight the critical parts of the communication routines. Accelerating the bottlenecks starts by modeling the hardware in untimed virtual prototypes and the software in a range of candidate embedded processors. A novel path from hardware prototypes to actual silicon allows rapid characterization of the accelerator in terms of power, performance and area. The suggested approach triggers a speedup from one order of magnitude in bottleneck components of Portals, while it is up to two orders of magnitude faster in both MPI and GA baseline implementations in a recent embedded processor.

Acknowledgements

I would like to thank my supervisor Dr. Ioannis Papaefstathiou and my industry supervisor Dr. Pavlos Mattheakis for the guidance, support, constructive remarks, devoted time, as well as the opportunities and challenges they presented me. Moreover, I would like to thank Professors Dionisios Pnevmatikatos and Apostolos Dollas for their co-advising and their participation in my Master Thesis committee. I would also like to thank Antonis Psathakis for his guidance in On-Chip SRAM power consumption. Furthermore I also thank the current members of Telecommunication Systems Institute (TSI) for their valuable advice, G. Chrisos, B. Savvakos, A. Brokalakis, K. Georgopoulos, Antonis Nikitakis, K. Miteloudi, P. Malakwnakis, C. Rousopoulos, S. Apostolakis, S. Nikolakaki and E. Strataki. Finally, I would like to thank my friends and family, my father George, my mother Irene,my brother Manos, as well as Xrysa for their love, support and encouragement.

This work was carried out with the financial support of Telecommunication Systems Institute (Chania Crete Greece) in the framework of the FASTCUDA, HERMES and R3-COP European programs with scientific support of Ioannis Papaefstathiou.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays there is renewed interest in high-performance computing community in parallel programming models due to the need to satisfy applications requirements with low power. Achieving an exascale level of performance requires several fundamental changes in hardware and software that will likely impact all areas of high-performance computing [1]. Hence, systems with numerous CPUs, along with extensive use of customized accelerators, have been recently proposed as, probably, the only viable solution offering high performance at a low energy budget [2].

Unfortunately, in highly parallel systems with a vast number of cores there are a lot of factors which can trigger significant system's underutilization. The major factor of them is communication, during which, cores may remain idle, waiting to synchronize or to exchange data to each other. Increasing the number of cores in a system also increases the execution time spent in communication, as each core has less work to do and usually more other cores to communicate with, because the problem is divided into more stages. Hence, a crucial challenge is communication's overhead reduction as the number of cores is increased.

Portals is an intermediate layer which intends to allow scalable, high-performance network communication between nodes of a parallel computing system. Portals is based on the concept of elementary building blocks that can be combined to support a wide variety of upper-level point-to-point and partitioned global address space (PGAS) network protocols [3]. Especially, the ultimate destination in memory of an incoming message is determined at the receiver by comparing contents of the message header with the contents of structures at the destination. This flexibility allows for efficient implementations of both one-sided and two-sided communications.

Portals protocol has been developed only in software until now by Sandia national laboratories. However, many communication protocols (such as MPI) today provide a simple mapping to hardware which may not be conducive to building upper layer protocols. The most works offloads the MPI communication tasks to co-processors, embedded processors on the NIC or even to dedicated hardware [4]. To the best of our knowledge, there isn't complete hardware implementation to offload the Portals building blocks and upper-level protocols which based on it.

The development of each new embedded platform is mainly comprised of two phases, the software and hardware one. While the software phase should be initiated as early as possible, the hardware phase needs several months or years. To reduce the product development time, virtual platforms are used giving to the software developers a virtual model of the hardware platform at the very earliest stage of the products development. Virtual models usually allows to instantiating an abundance of popular architectures (such as MIPS, ARM, PowerPC) along with widely used peripheral models, such as USB, DMA controllers etc. The above processors with their peripherals can be co-simulated with the under-development hardware models with respect to accuracy of most significant architecture parts and simulation speed.

This work presents Open Virtual Platform (OVP) framework for exploring accelerator-based architectures for multiprocessor systems. This framework allows for the instantiation of processor models along with under-development accelerators, so that architecture parameters can be evaluated with realistic software and not with corner testcases and synthetic benchmarks. Moreover, and more importantly, this work presents certain novel Portals acceleration core using the existing OVP framework.

Subsequently, a hardware implementation flow is inserted creating the cycle accurate model of our Portals processor using Cadence C-to-silicon high level synthesis tool. Moreover, a set of rules of thumb are presented, guiding the transformation of untimed functional descriptions to cycle accurate ones. Then, using the C-to-silicon high level synthesis tool, the cycle accurate descriptions are optimized and synthesized with a modern standard cell technology library.

Finally, to demonstrate the effectiveness and accuracy of our work, certain tasks of Message Passing Interface (point-to-point) and Global Arrays (PGAS) [5] upper layer Protocols are implemented using Portals Routines within a number of widely used MPI and GA benchmarks.

This master's thesis is organized as follows. Chapter 2 introduces an overview of the Portals API and basic operations of this. Chapter 3, initially presents a framework allowing for the rapid exploration of novel parallel architectures when executing real-world applications and Portals Accelerator integration using existing framework. Subsequently the same Chapter describes the Portals routines which are implemented as well as the architectures for offloading certain Portals tasks. Chapter 4 presents the transformation from virtual accelerator to silicon using state-of-the art synthesis tool as well as multithread Platform integration with our H/W Portals Accelerator. Chapter 5 presents our real-world experimental results, based on different MPI and GA parallel benchmarks. Finally, in Chapter 6, conclusion and future work are presented.

2

An overview of the Portals API

Portals intermediate layer has been developed for nearly twenty years by Sandia National Laboratories and the University of New Mexico and it is intended to allow scalable, high-performance network communication between nodes of a parallel computing system. This chapter initially introduces the concepts of one-sided, two-sided, blocking and non-blocking communication operation in Portals protocol. Afterwards, it presents an overview of the Portals basic data movements. Subsequently, it demonstrates the operation of Portals Lists, completion event mechanism, rendezvous protocols as well as memory descriptors definition. At the end, matching and non-matching Network Interface is analyzed as well as lists operations of data movements are described in details.

2.1 One Sided versus Two-Sided Communication

In two-sided communication both sender and receiver require an implicit synchronization where the messages are matched using a combination of tag (message identifier). In other words, both sender and receiver must issue one communication routine so as to exchange one Data Packet. On the other hand, in one-sided communication, one process accesses the remote memory of another process directly without interrupting the progress of the later process. Using this programming model could reduce the punitive synchronization costs of multi-core machines, compared to two-sided communication. Portals provide both two-sided and one-sided data movement operations, but unlike other one-sided programming interfaces, the target of a remote operation is not a virtual address. Instead, the ultimate destination in memory of an incoming message is determined at the receiver by comparing contents of the message header with the contents of structures at the destination. This flexibility allows for efficient implementations of both one-sided and two-sided communications. In particular, Portals is aimed at providing the fundamental operations necessary to support a high-performance and scalable implementation of the Message Passing Interface (MPI) standard and Partitioned Global Address Space (PGAS) Models.

2.2 Two-sided non-blocking versus blocking operation

As described in chapter 1, the communication can consume a huge part of the run time of a parallel application. So, the communication time in those applications can be addressed as overhead because it does not progress the solution of the problem in most cases. Using overlapping techniques enables the user to move communication and the necessary synchronization in the background and use parts of the original communication time to perform useful computation. Figure 2.1 illustrates an abstract view of two processor distributed memory system when exchanging one message. Processor P1 executes one point-to-point send command, while processor P2 executes one receive command so as to accept message.

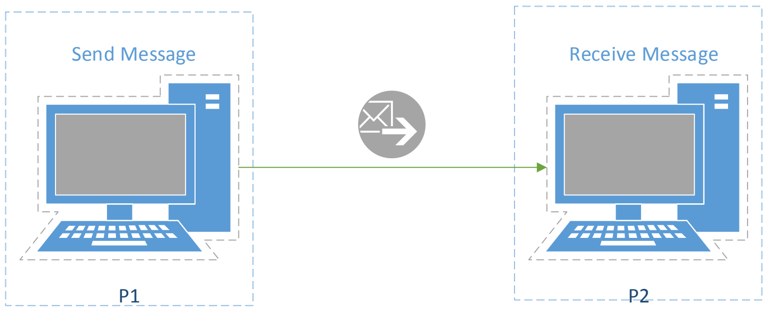


Figure 2.1: Send-Receive point-to-point Operation

There are two scenarios which P2 can accept the message as depicted in Figure 2.2. The send command is non-blocking which means that P1 is able to resume its computation right after message is dispatched. On the other side, P2 either executes a blocking receive operation, or follows a non-blocking receive scenario. In blocking case, which is depicted in Figure 2.2a, if the message has not yet arrived when P2 executes blocking receive command, P2 must remain in blocked state until the actual message finally arrives. Otherwise, if the message has already reached in P2 when the receive command is executed, it would be delivered immediately to P2 user space, and the computation would be resumed. On the other hand, in non-blocking scenario P2 returns instantly to computation$_b$ when it executes a non-blocking receive command. So, at some point, P2 needs the data contained in the message and thus waits for the message to arrive by executing a blocking wait command. As soon as the message arrives in P2, blocking wait command returns to computation$_a$ indicating the successful arrival of data. To sum up, in case of blocking operation P2 remains idle more time than non-blocking operation; so the later must be used from programmers to overlap the communication using independent computation routines.

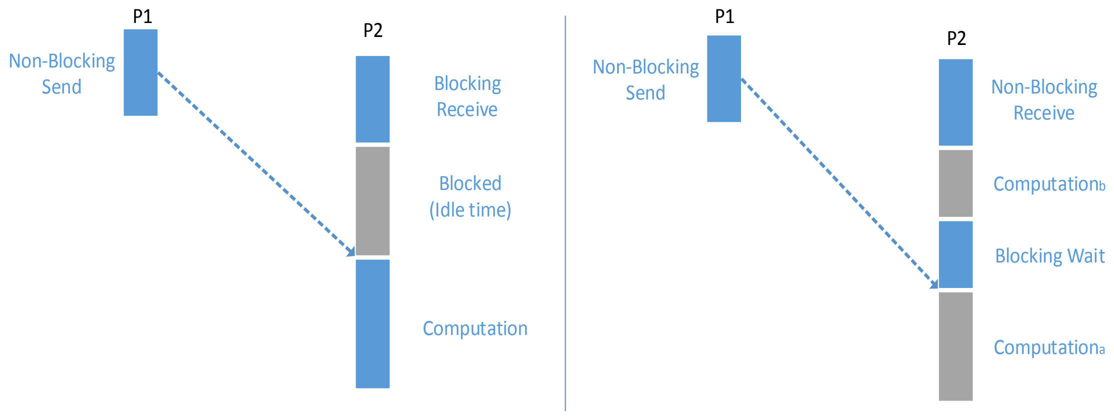


Figure 2.2: (a) Blocking Receive (b) Non Blocking Receive commands

2.3 An overview of Portals Data Movements

Portals use two basic data movements operations Ptl_Put and Ptl_Get. In case of Ptl_Put operation, the sender sends one (or more) Data Packet(s) to receiver, while in Ptl_Get operation one node (which requires data from some other node) sends one Header (Ptl_Get) to other node waiting for response. Another Portals movement operation category is Ptl_Atomic (a combination of Ptl_Put and Ptl_Get) operation. Especially, it can be either a swap operation which is used for data exchange or an accumulate operation which combines the incoming data with the data that resides at receiver process.

Every data movement operation involves two processes (nodes), the **initiator** (Sender) and the **target** (Receiver). In other words, the initiator is the process that initiates the data movement operation, while the target is the process that responds to the operation by accepting the data for a put operation, replying with the data for a get operation, or updating a memory location for, and potentially responding with the result from, an atomic operation.

Another considerable Portals routine is the Ptl_Append routine. Ptl_Append is used from target node when it wants to receive (in Ptl_Put) or send (in Ptl_Get) message data for two-sided communication. In other words, with this routine target responds to the operation (such as MPI Receive Command). Figure 2.3 illustrates an abstract view of Portals Put, Get and Swap Atomic operations.

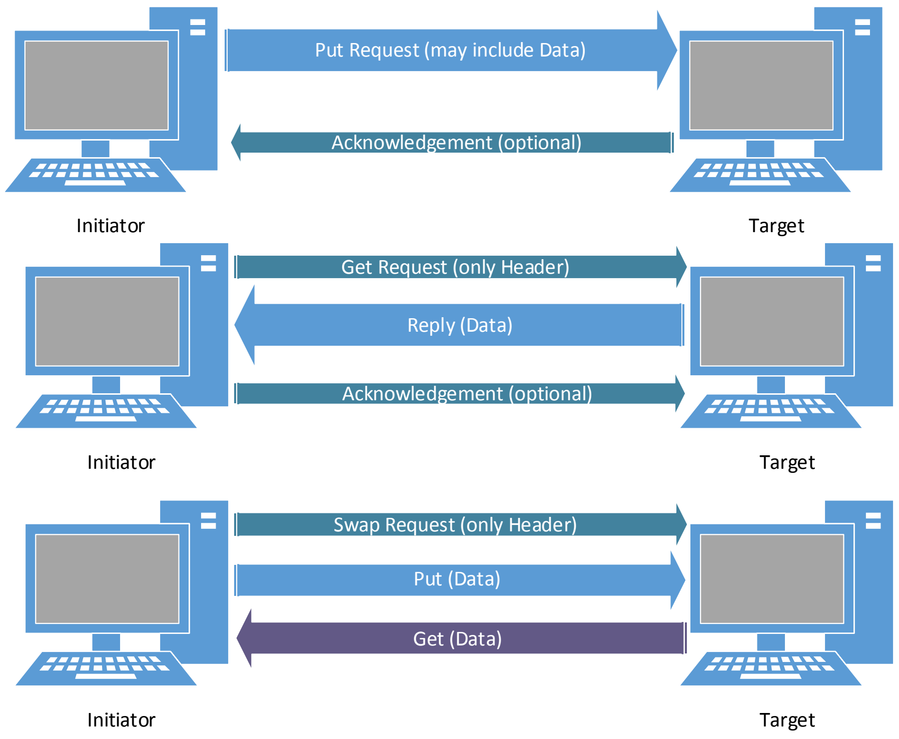

Put Request (may include Data)
Acknowledgement (optional)
Initiator
Target
Get Request (only Header)
Reply (Data)
Acknowledgement (optional)
Initiator
Target
Swap Request (only Header)
Put (Data)
Get (Data)
Initiator
Target

Figure 2.3: (a) Put (b) Get (c) Swap Atomic Operation

2.4 Portals Lists

Portals uses three lists at target side so as to manage its messages: Priority, Unexpected and Overflow List. Priority List stores the header of the expected messages through Ptl_Append operation, Unexpected List stores the headers of unexpected messages, while Overflow List stores the data of small unexpected messages as illustrated in Figure 2.4.

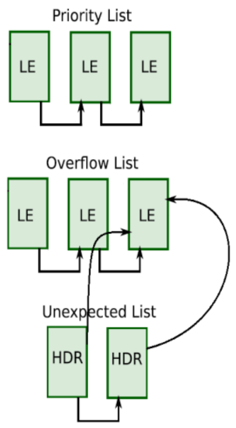


Figure 2.4: Portals Target Lists

UM and OF Lists grow with programs which use eager put commands and in which the sender assumes that the receiver has enough space to buffer the header and payload are sent respectively, while PR List grows with target receive commands. In other words, at message arrival the priority list are first processed and, if no matching entry was found, then overflow list are processed to find available space for message payload. If there is available space, a message payload is delivered into the overflow list and its header is linked into the unexpected list. On the other hand, when a new list entry is appended to the priority list, the unexpected list is first searched for a match. If a match is found (i.e., had the list entry been on the priority list when the message arrived, the message would have been delivered into that list entry), the list entry is not inserted, the header is removed from the unexpected list, and the application is notified a match was found in the unexpected list.

Figure 2.5 illustrates how to grow lists in target side through an example. We assume that we have four communication nodes in a system. We examine the target side when node 1, node 2 and node 3 send messages (initiators) to Node 0 (target). Initially, in Portals initialization function, node 0 issues *PtlAppend(PTL_OVERFLOW_LIST)* so as to allocate space for unexpected messages (Figure 2.5a). Subsequently, node 0 wants to receive one message from node 1, hence it issues *PtlAppend (PTL_PRIORITY_ LIST,1) so* as to insert one entry in Priority List as shown in Figure 2.5b. Later, two incoming messages arrive in node 0 from nodes 2 and 3 (Figure 2.5c and Figure 2.5d respectively), as a result two headers[1] in UM List are inserted, since the node 0 don't wait messages from those nodes. Then, node 0 issues *PtlAppend(PTL_PRIORITY_LIST,2)* command when it wants to receive the corresponding message (Figure 2.5e), as a result corresponding UM entry is deleted. Finally, the expected message from node 1 arrives (Figure 2.5f) and node 0 issues *PtlAppend (PTL_PRIORITY_LIST,3)* for the last message from node 3 (Figure 2.5g).

[1] When one unexpected message arrived in a target node, header is saved in unexpected message list, while payload is saved in Overflow List.

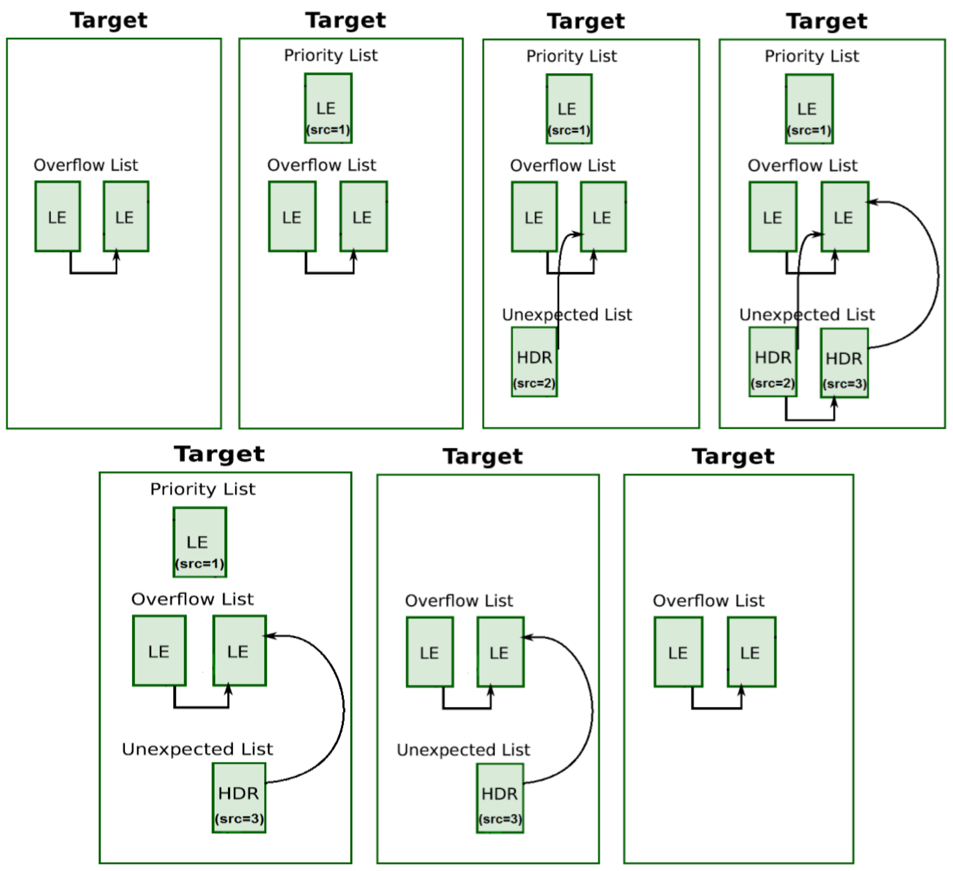


Figure 2.5: Simple example of Target Lists

Unfortunately, the size of three Lists affects Portal's communication overhead, as applications may traverse a significant number of entries searching for a certain message. Hence, accelerating queue as well as other communication operations can result to reduced Portals overhead. Figure 2.6a describes in more detail the Ptl_Put operation executed by P2 during a non-blocking[2] receive comparing with Figure 2.2. In the abstract view (Figure 2.2) it was assumed that a non-blocking receive call returns as soon as the receive is posted. Looking at the scenario in more detail a number of operations should be performed in order to execute fully receive command. When the target wants to receive one message from other node it calls Ptl_Append routine so as to append a receive request. During this process it must first search in UM List in case of message has received in the past (Process UM). Accordingly, when the message arrives in P2 the computation is not instantly resumed, since the Priority list has to be traversed in order to check whether a matching receive has been posted in the past (Process PR). If there isn't any matching receive in Priority List, it traverses OF List to find sufficient space to save the unexpected message (Process OF). Ptl_CT_Wait is one blocking command which waits for the message arrival. In Figure 2.6a UM, PR, OF Process

[2] All Data movements in Portals are non-blocking (such as Ptl_Put, Ptl_Append) except from Ptl_CT_Wait.

and CT_Wait have to be performed by the host processor, incurring a significant overhead in the case of a large queues. Offloading all communication intensive processes from the host processor to an accelerator allows for extensive overlap between the computation-communication without further involvement as illustrated in Figure 2.6b.

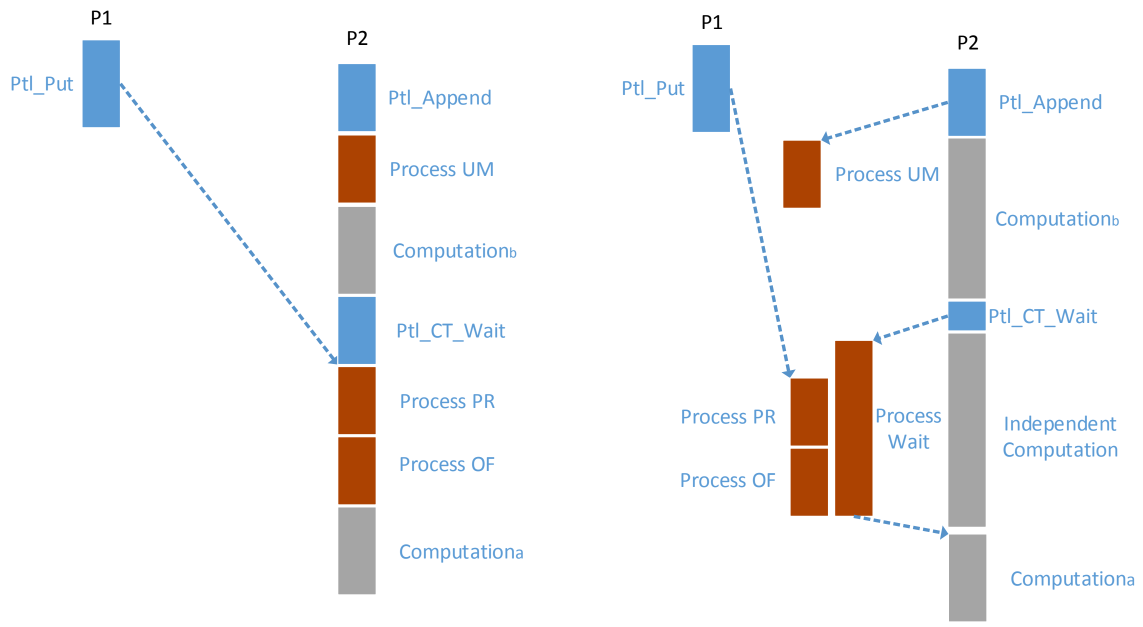


Figure 2.6: (a) CPU non-Blocking Receive (b) Offload Non Blocking Receive schemes

2.5 Eager versus Rendezvous Protocol

Historically, two-sided communication implementations have had to choose between eager messaging protocols that require buffering and rendezvous protocols that sacrifice overlap and strong independent progress in some scenarios [7]. The typical choice is to use an eager protocol for short messages and switch to a rendezvous protocol for long messages. This subsection presents both eager and rendezvous protocols.

The eager protocol sends whole messages including Header and Payload eagerly as illustrated in Figure 2.7. If a message is expected[3], it is delivered directly into the userSpace Buffer and an ack (optionally) is generated to notify the sender that the message was successfully delivered (Figure 2.7a). On the other hand, if the message is unexpected, there are two probable scenarios. In the first scenario (Figure 2.7b) the receiver discards the payload of the message while it keeps the Header so as to issue a get request to retrieve the payload when PtlAppend command is issued. In the second scenario, target node has sufficient bounce buffer to save the payload so that initiator doesn't send the payload second time as illustrated in Figure 2.7c.

[3] When PtlAppend has been issued by target node before the message arrives the message is expected, else the message is unexpected.

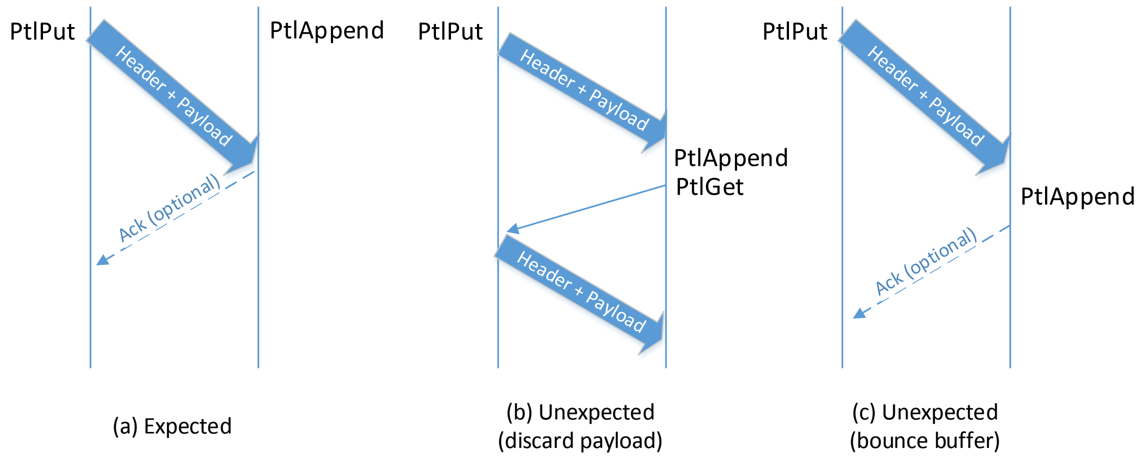


Figure 2.7: Communication pattern for eager protocol

In case of Rendezvous protocol, initiator only sends a piece of message[4] and sufficient information in the header to allow the target to issue a get operation to retrieve the message when the PtlAppend command is posted as illustrated in Figure 2.8. If the message is expected the first part of the message is delivered directly into the receive buffer, otherwise it is delivered into bounce buffers. If the total size of message is greater than eager_limit, target issues PtlGet so as to retrieve the remainder payload.

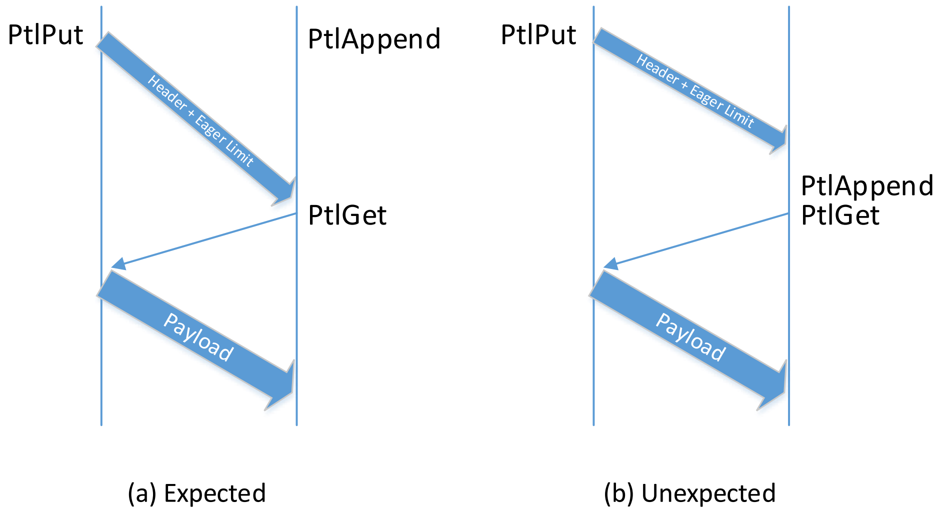


Figure 2.8: Communication pattern for rendezvous protocol

[4] The size of message is determined from one variable, called eager_limit.

2.6 Portals Completion Events

Portals provides two mechanisms for recording completion events, full events and counting events. Full events provide a complete picture of the transaction, including what type of event occurred, which buffer was manipulated, and identifying any errors that occurred. Counting events, on the other hand, are designed to be lightweight and provide only a count of successful and failed operations (or successful bytes delivered). The delivery of events (full events or counting events) is manipulated when creating a number of other structures (such as creation of priority list). In this work counting events are implemented in Portals Accelerator because it was a lighter edition of Portals events with sufficient information of transactions. A counting event is allocated through a call to PtlCTAlloc(), queried with PtlCTGet(), PtlCTWait(), set with PtlCTSet(), incremented with PtlCTInc(), and freed through a call to PtlCTFree().

2.7 Portals Memory Descriptors

A memory descriptor contains information about a region of a process[5] memory and optionally points to an event queue or counting event where information about the operations performed on the memory descriptor are recorded. Memory descriptors are initiator side resources that are used to encapsulate the association of a network interface (NI) with a description of a memory region. They provide an interface to register memory and to carry that information across multiple operations (an MD is persistent until released). PtlMDBind() is used to create a memory descriptor and PtlMDRelease() is used to unlink and release the resources associated with a memory descriptor.

2.8 Matching & non-Matching Network Interface

The Portals API supports the use of two network interfaces, physical network interface and logical network interface. A Portals physical network interface is a per-process abstraction of a physical network interface, while a logical network interface associated with a single physical network interface share the same network id and process id (nid/pid), but all other resources are unique to a logical network interface. Logical network interfaces may be matching or non-matching and can be addressed by either logical (rank) or physical (nid/pid) identifiers. In this work logical network interface is used which is addressed by logical (rank) identifiers for simplicity reasons.

In non-matching Portals interface, the initiator needs to send (i) number of target node (rank number), (ii) number of initiator node, (iii) Portal's Operation, (iv) length of Payload,(v)

[5] A memory descriptor describes a memory region using a base address and length.

remote offset, and (optional) other header data. Figure 2.9 illustrates a synchronous[6] Portals Get Operation using non-matching NI, as a result the first list entry (LE) in a list always matches. In addition to the standard address components, in matching network interface (Figure 2.10) a portals address additionally includes a set of match bits. For a matching logical network interface, each match list entry specifies two bit patterns: a set of "do not care" bits (ignore bits) and a set of "must match" bits (match bits). Along with the source node ID (NID), these bits are used in a matching function to select the correct match list entry. Incoming match bits are compared to the match bits stored in the match list entry using the ignore bits as a mask. An optimized version of this is shown in the Figure 2.11.

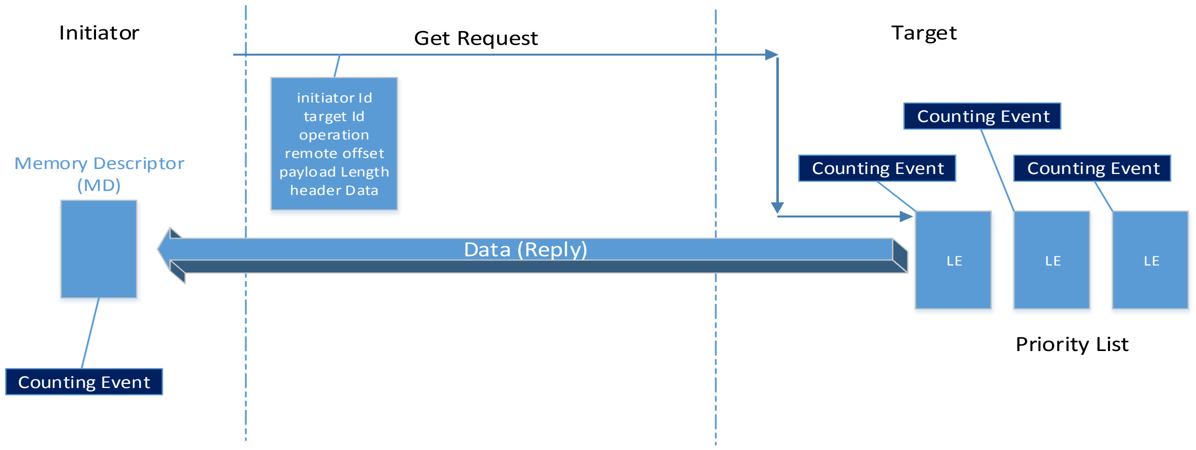


Figure 2.9: Synchronous Portals Get from a list entry

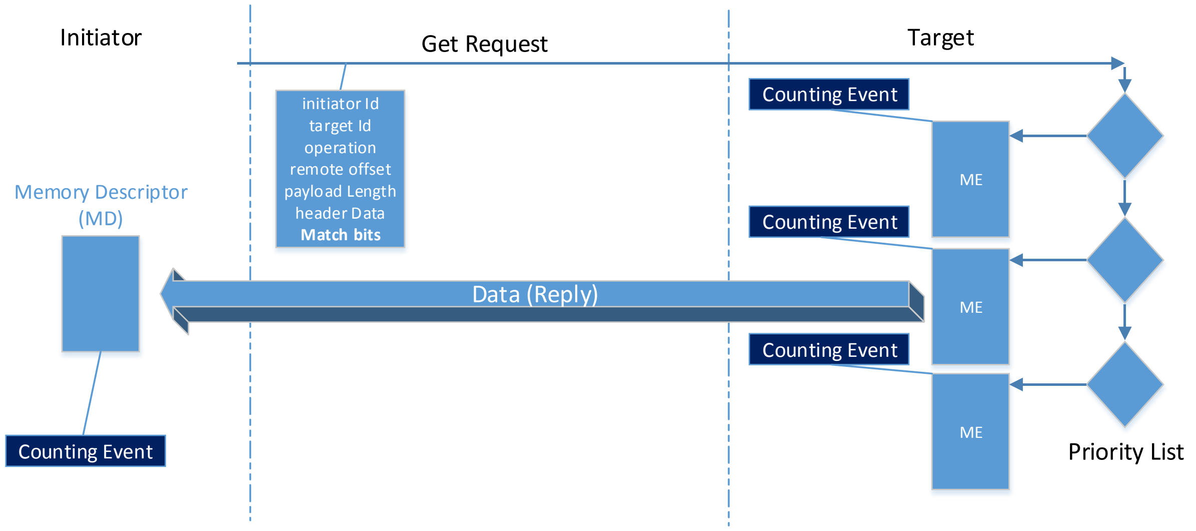


Figure 2.10: Synchronous Portals Get from a match list entry

[6] This transaction is synchronous because the target has been issued a Ptl_Append(PTL_PRIORITY_LIST) before the initiators get request.

```
((incoming_bits ^ match_bits) & ~ignore_bits) == 0
```

Figure 2.11: Portals Matching Function

2.9 Portals Data movements in detail

This section describes the details of Portals Data movements, the Portals lists operation of each transaction, and finally it shows the basic PtlPut transaction using C code as example.

2.9.1PtlAppend

The PtlAppend() function creates a single match list entry which is specified by ptl_list. Especially, ptl_list can be either PTL_PRIORITY_LIST or PTL_OVERFLOW_LIST and the corresponding entry is appended to the end of the appropriate list specified by ptl_list as illustrated in Figure 2.12.

In case of PTL_PRIORITY_LIST, when a match list entry is posted to the priority list, the unexpected list is searched to see if a matching message has been delivered in the unexpected list prior to the posting of the match list entry. If so, an appropriate overflow list entry passes the data to user space, the matching header is removed from the unexpected list, and a match list entry is not inserted into the priority list. On the other hand, if the message not found in Unexpected Message List, then a match list entry is posted to the priority list.

In case of PTL_OVERFLOW_LIST, the target inserts an Overflow List entry so as to save the Unexpected Messages in the future.

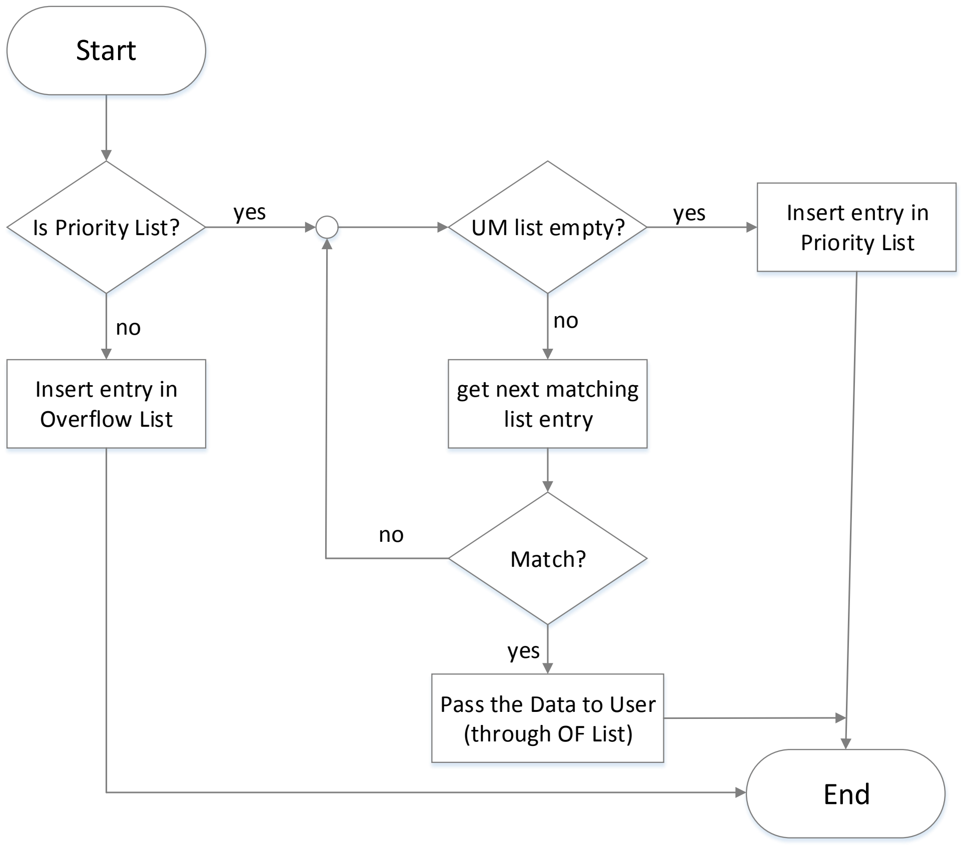

Start
Is Priority List?
yes
no
Insert entry in Overflow List
UM list empty?
yes
Insert entry in Priority List
no
get next matching list entry
Match?
no
yes
Pass the Data to User (through OF List)
End

Figure 2.12: Target manipulation of Portals PTLAppend function

2.9.2PtlPut

The PtlPut() function initiates a non-blocking[7] put operation as illustrated in Figure 2.13. When a message arrives in target side, the priority list is searched to see if a matching message has been posted beforehand. If so, the payload of the message passes immediately to the user space, and a match list entry is not inserted into the unexpected list. Otherwise, if the message not found in Priority List, the overflow list is searched to see if there is sufficient space to save the unexpected payload to the bounce buffer. If so, the payload of message is stored in Overflow List, while a match list entry is posted to the unexpected message list. Otherwise, the target discards the message and increments the drop counter.

[7] In Portals there isn't blocking put operation (such as MPI_Send in MPI two-sided protocol).

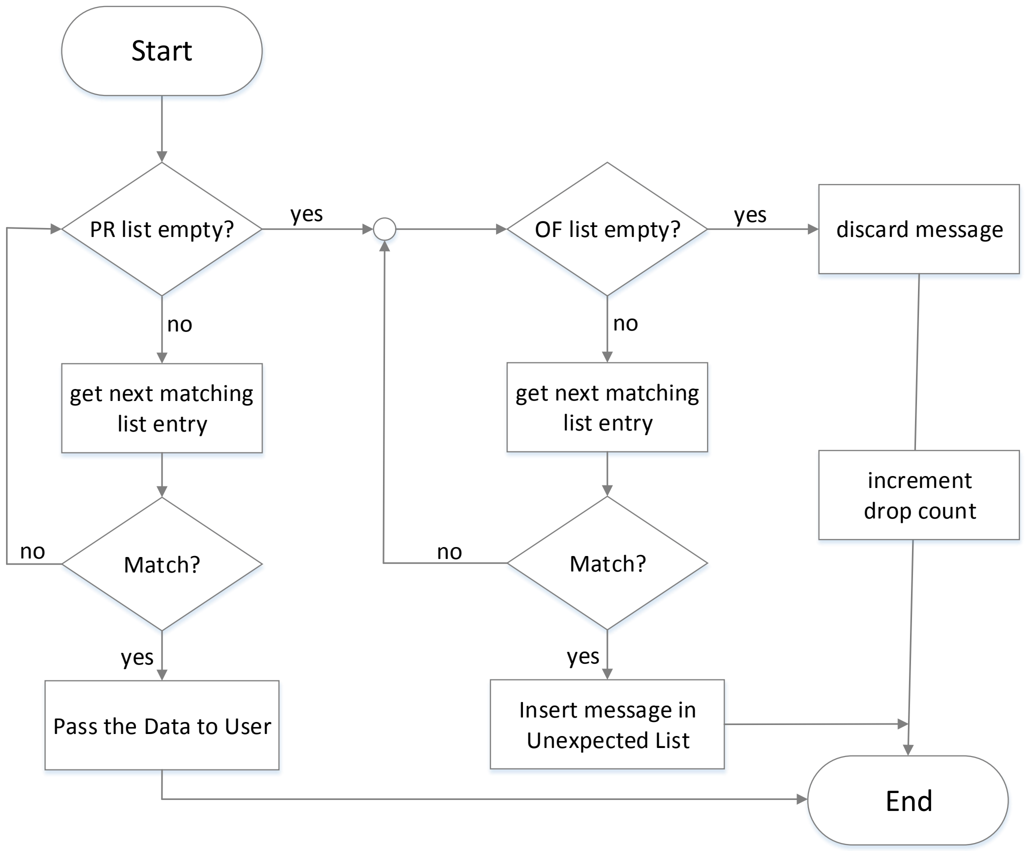


Figure 2.13: Target manipulation of Portals PTLPut function

2.9.3 PtlGet

The PtlGet function initiates a non-blocking remote read operation. In PtlGet transaction the target does not use an Overflow List entry, because it doesn't send Payload (just the message header) as a result there aren't Unexpected Payloads. When a get header arrives in target side (Figure 2.14), the priority list is searched to see if a matching message has been posted in the priority list prior to the header arrival. If so, the target immediately sends the payload of message back to initiator. Otherwise, if the message not found in the Priority List, a match list entry is posted to the unexpected message list with the header of the get operation.

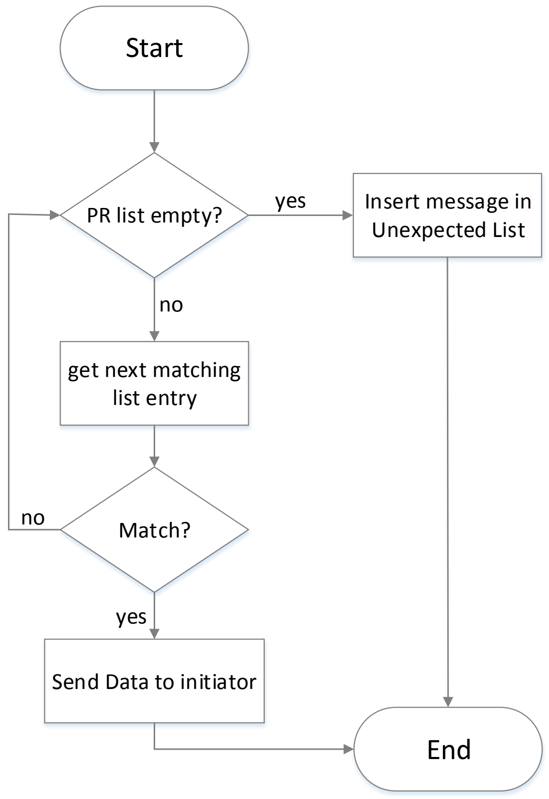

Start
PR list empty?
yes
Insert message in Unexpected List
no
get next matching list entry
no
Match?
yes
Send Data to initiator
End

Figure 2.14: Target manipulation of Portals PTLGet function

2.9.4 Portals PtlPut Two-Sided Communication Example

The following example shows the PtlPut two-sided Portals transaction using matching network interface as described in Portals 4 Library [8]. Initially, target allocates buffer space to store the unexpected messages with PtlMEAppend function. In addition, all nodes must be synchronized so as to ensure the bounce buffer creation. Subsequently, node 1 (initiator) initializes the MD structure and binds a MD entry. On the other side, node 0 (target) initializes the ME entry structure, allocates one CT entry, and issues PtlMEAppend(PTL_PRIORITY_LIST) so as to accept the message. Then initiator selects the target node, sends the message (through PtlPut) and releases MD. Finally, target node waits for the message arrival, checks the success counter, and frees the CT entry.

Example Two-Sided Send-Receive transaction in Portals

```
#define BUFSIZE 4096
if(rank == 0){ /* Create OF List entry in target side */
   unexpected_e.length = BUFSIZE; /* Size of unexpected entry */
   PtlMEAppend(&unexpected_e, PTL_OVERFLOW_LIST, &unexp_handle);
}
Barrier(); /* Synchronize all nodes */

if(rank == 1){ /* Bind space and store the data */
   send= 15; /* Send the value 15 to target */
   write_md.start  = &send;
   write_md.length = sizeof(Uns32);
   PtlMDBind(&write_md,&write_md_handle);
}

if(rank == 0){
   value_e.start         = &rcv;
   value_e.length        = sizeof(Uns32);
   value_e.match_id.rank = 1; /* Source */
   value_e.match_bits    = 1;
   value_e.ignore_bits   = 0;
   PtlCTAlloc(ni_h, &value_e.ct_handle);
   PtlMEAppend(&value_e, PTL_PRIORITY_LIST,&value_e_handle);
}

if(rank == 1){
   ptl_process_t peer;
   peer.rank = 0; /* Send the DP to 0 target node */
   Uns32 match_bits = 1;
   PtlPut(write_md_handle, sizeof(Uns32), peer, match_bits);
   PtlMDRelease(write_md_handle);
}

if(rank == 0){ /* Read value in target side */
   PtlCTWait(value_e.ct_handle, 1, &ctc);
   printf("success %d\n",(Uns32)ctc.success);
   printf("receive value %d\n",rcv);
   PtlCTFree(value_e.ct_handle);
}
```

3

Software-driven development of a mixed Software & Hardware Portals System

This chapter introduces a software framework for exploring accelerators' architectures within a multi-parallel system while mainly focusing at the exploration of our novel Portals Accelerator. The presented framework is based on the virtual platform simulation environment OVP. Initially, it analyzes our novel Portals Accelerator implementation and its intercommunication using the OVP environment, and finally it presents Rendezvous protocol and One-Sided communication implementations.

3.1 Introduction to Virtual Platform

The development of each new embedded platform is mainly comprised of two phases, the software and hardware one. While the software phase should be initiated as early as possible, the hardware phase needs several months or years. The most common practice for developing embedded software is to start to develop initial software in a desktop running a general purpose operating system before the real hardware or prototype release. When a prototype of the embedded system or chip is available the software is ported to this target environment using cross compilers and related tools. Later, when final Hardware is available, further modifications are needed for the product release. There are many challenges when using this traditional approach. Initially, there are significant differences in host-based and target hardware environment. Moreover, hardware prototype is often physically unreliable, not readily available in developments sites (especially those off-shore), and worst of all, it is often available only very near to the end of the targeted product development schedule. These challenges become acute as more processors interact in the embedded system. One solution is the MPSoC Software developing using separate processors which are emulated by distinct host threads. This approach provides limited controllability, observability, and debugability especially when tracking down complex multi-processor issues, such as bugs which are often very hard to reproduce reliably and isolate in complex real-time hardware.

The usage of Open Virtual Platform (OVP) [9] is to reduce the product development time by months, especially for MPSoC platforms, giving to the software developers a virtual model of the hardware platform at the very earliest stage of the products development. OVP usually allows to instantiating an abundance of popular architectures (such as MIPS, ARM, PowerPC) along with widely used peripheral models, such as USB, DMA controllers etc. The above processors with their peripherals can be co-simulated with the under-development hardware models with respect to accuracy of most significant architecture parts and simulation speed.

OVP advantages would have been useless without an accurate efficient simulator. For this reason, Imperas which initiated the OVP platform, has made available the OVPsim simulation tool. OVPsim provides infrastructure for describing platforms with one or more processors containing shared memory and busses in arbitrary topologies and peripheral models. Performance of OVPsim depends on several factors (such as the processor variants used in the platform, the exact nature of the application itself), but typically estimation is hundreds of millions of simulated instructions per second. Finally, OVPsim provides the ability to hook up to any popular external debugger that supports the GNU protocol, such as GDB.

3.2 Portals Accelerator using Imperas OVP

This section introduces a framework for exploring accelerator's architecture and especially presents the implementation of our Portals Accelerator using OVPsim. In this thesis, the Open

Risc 1000 (OR1K) processor model was selected to be the computation core of each node, while the same design process can be followed in case of different process models (such as ARM, Power PC etc.) since the presented results are independent with the processor model. Innovative CPU Manager (ICM) API is used for OR1K initialization and instantiation. Moreover, certain functions (such as icmPrintf) can performed at the host machine and not at the simulated platform. This feature along with the ability to hook the execution software running on a processor directly providing the GDB debugger can be reduce significantly the development process in a multiprocessor platform. During system's simulation, each of the OR1K processors is executed in the host machine at predefined time slices. The scheduler selects one processor and simulated it for one time slice. Especially, the simulator calculates the number of instructions that should be executed by that processor in a time slice, and then simulating for that number of instructions. When this processor has simulated for a time slice, it is suspended and the next processor is simulated for the same time. This is a pseudo-parallel approach which emulates the concurrent behavior of an actual multi-processor platform. We get the better trade-off between simulation and accuracy of results using time slice of 1ms.

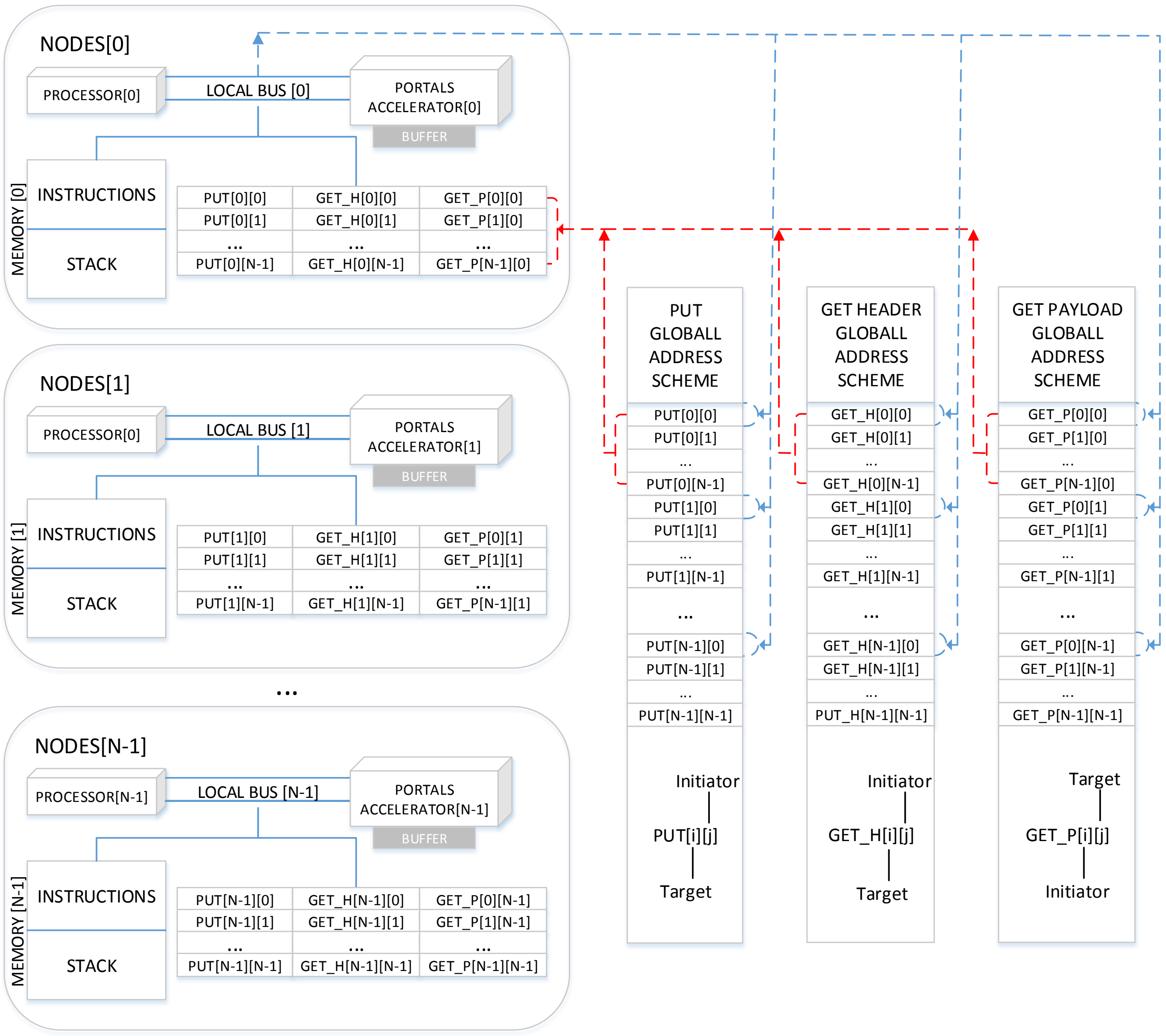


Figure 3.1: OVP Platform with our Portals Accelerator

An overview of Platform which is used in this work is shown in Figure 3.1. The platform consists of N processing nodes and each node comprised of one OR1K processor model, its memory, and one Portals Accelerator connected in a local bus. In turn, Portals Accelerator comprised of a dynamic memory allocator peripheral for the unexpected received Portals messages, a list manager for its queue processing and an Accelerator Buffer. Furthermore, Figure 3.1 shows the space reserved for the memory mapped intercommunication. The field which corresponds to the Portals accelerators is consistent among all processors because one global address scheme is used for PtlPut and two global address schemes for PtlGet transaction. In PtlPut transaction, initiator node writes the message to the appropriate global space in *'PUT GLOBAL ADRESS SCHEME'*, while target node can read the message from the same space. In PtlGet transaction, when the initiator node writes the header of the message request in *'GET HEADER GLOBAL ADRESS SCHEME'*, the target reads the header from the same space. Later on, the target, in turn, responds to initiator request with the message Payload using the *'GET PAYLOAD GLOBAL ADRESS SCHEME'*. In other word, there is a unique global address which associates an address interval with the Portals buffer of a specific node. We use three global address schemes because one node may issue PtlPut and PtlGet operation concurrently to some other node (i.e. in Rendezvous Protocol).

The memory of each node consists of two parts, instructions and stack as illustrated in Figure 3.1. Those parts, along with the space reserved for the memory mapped intercommunication as well as the accelerator buffer memory are shown in Table 3.1. The size of *'PUT GLOBAL ADRESS SCHEME'* is determined form following equation.

$$PUTGlobalSize = NPROCS^2 * PortalsMessageSize - 1 \tag{1}$$

In this work, up to 128 Software processors are instantiated with maximum PortalsMessageSize 8192 bytes for our results. With this configure the PUTGlobalSize has size 07ffffff bytes. Similar to (1), the GetHeaderGlobalSize is computed with the following equation.

$$GetHeaderGlobalSize = NPROCS^2 * PortalsHeaderSize - 1 \tag{2}$$

As described in Chapter 2, Portals Header contains Source, Destination, LocalOffset, RemoteOffset, MessageLength integers, PayloadStart pointer and MatchBits with Uns64. In other words, PortalsHeaderSize and GetHeaderGlobalSize should be at least 32 and 7ffff bytes respectively. Hence, fffff bytes are sufficient for *'GET HEADER GLOBAL ADDRESS SPACE'*. The remainder range (from 0x60100000 to 0x6fffffff) is assigned for *'GET PAYLOAD GLOBAL ADDRESS SPACE'* which are sufficient according to (1) equation. Finally, range from 0x70000000 to 0x7fffffff is assigned for dynamic memory allocator buffer, while ListManager and other accelerator structures allocate their buffer in range 0x58000000 up to 0x5ffffff.

Address		Size	Mapped
Low	High		
0x00000000	0x0fffffff	0fffffff	Instructions
0x90000000	0xffffffff	6fffffff	Stack
0x50000000	0x57ffffff	07fffff	PUT GLOBAL ADRESS SHEME
0x58000000	0x5ffffff	07fffff	Portals Accelerator Buffers
0x60000000	0x600fffff	000fffff	GET HEADER GLOBAL ADDRESS SPACE
0x60100000	0x6fffffff	0feffff	GET PAYLOAD GLOBAL ADDRESS SPACE
0x70000000	0x7fffffff	0fffffff	Dynamic Memory Allocator Buffer

Table 3.1: Memory Mapping

The Portals Accelerator is modeled by the OVP Innovative CPU Manager (ICM) API and configured by the Peripheral Programming Model (PPM) as outlined in detailed in the next section. Moreover, Behavioral Modeling (BHM) API allows passing parameters to the peripheral during instantiation. This feature was used to assign an ID to each accelerator, so that the address interval of the local Portals buffer can be identified. For instance, assigning ID = 1 to an accelerator in a system with 128 nodes, resulted to assigning to the local buffer the global address interval of [LowGlobalAddress+1*128*PortalsMessageSize : LowGlobalAddress+2*128* Portals MessageSize-1] for PtlPut and PtlGet response (Payload) operation. Similar to the above operations, for the same ID=1, the global address interval of PtlGet Header transaction is [LowGlobalAddress+1*128*PortalsHeaderSize : LowGlobalAddress +2*128*PortalsHeaderSize-1]. Although the above memory mapping can be assigned during the initialization phase of platform, OVPsim requires to perform this task in the instantiation phase in order to assign certain permissions to each address as described in the following section. Whenever one message was written at the local buffer of Portals Accelerator, it is consumed immediately by the appropriate target's accelerator preventing buffer overwrites.

3.3 Accelerator Intercommunication

This section presents the bus connections as well as accelerator intercommunication with other nodes, its buffer and UserBuffer. The bus connects the platform nodes to a common address space so that all processor read/write commands are directed to Portals Accelerator Buffers. Portals Accelerator connects to the bus through master and slave ports depending on whether it creates or responds to bus transactions as described in next subsections in detail.

3.3.1 Master Ports

This subsection describes the master ports which connect the Accelerator with node's buffer as well as with other nodes. Especially, the accelerator uses a master port to initialize bus transactions to read/write to the memory. Figure 3.2 illustrates the Accelerator's master ports which are created in this work. Accelerator in our platform uses five master ports to communicate with the local bus. One master port (1 in Figure 3.2) is used to read/write the node's UserSpace, so that the processor is not interrupted in any way. Another master port (5 in Figure 3.2) is used to store/restore unexpected messages payload in allocator buffer, while three master ports (2-4 in Figure 3.2) are allocated to communicate (read/write Portals messages) with other nodes, one for each Global Address Scheme.

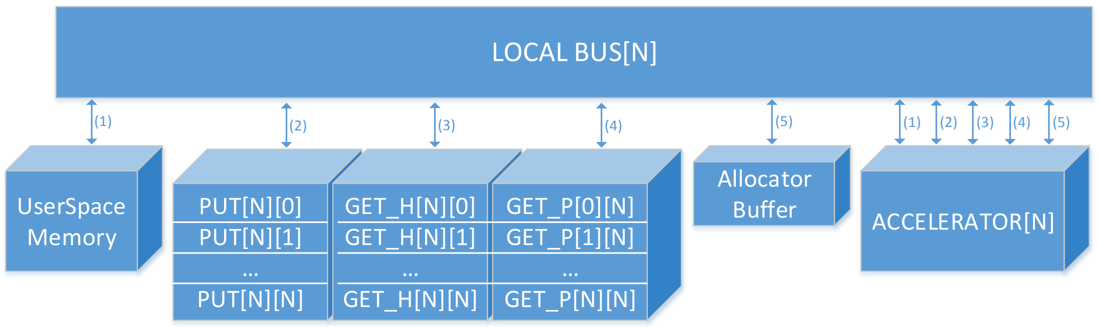


Figure 3.2: Accelerator's master ports

Innovative CPU Manager (ICM) API is used to create Global Address Schemes and connect them to local bus. Code 1 describes the PUT_GLOBAL_ADDRESS_SCHEME creation and how it is connected with the local bus. Initially, we select the Base Address according to Table 3.1. Subsequently, we create a new Global memory with Read-Write privilege and size $NPROCS^2 * PortalsMessageSize - 1$ with **icmNewMemory** OVP command, while we connect it to each accelerator bus with **icmConnectMemoryToBus**. Similar code has been used for the other two Global Memories, while it can be used for any other Global memory instantiation using Imperas OVPsim.

Code 1 Create and Connect Global memory to bus

```
#define PUT_GLOBAL_BASE 0x50000000 //* Base Address *//
//* Bind Space for PUT GLOBAL ADDRESS SCHEME *//
icmMemoryP PutMsgMemory = icmNewMemory("PutGlobalMem",ICM_PRIV_RW,
                          NPROCS*NPROCS*PTL_MESSAGE_SIZE-1)
//* Connect PUT GLOBAL ADDRESS SCHEME to Accelerator buses  *//
for AccId=0,...,NPROCS:
  icmConnectMemoryToBus(LocalBus[AccId], PutMsgMemory,
                       PUT_GLOBAL_BASE)

```

Code 2 Master Ports Operations

```
//* Master ports Handler *//
ppmAddressSpaceHandle msgPayloadHandle[NPROCS]
for i=0,...,NPROCS: //* Open NPROCS master ports to each Acc *//
  msgPayloadHandle[i] = ppmOpenAddressSpace(PortNames[i])
  //* Connect each master port to local bus *//
  icmConnectPSEBus(Acc[AccId], LocalBus[i], PortNames[i])

//* Select the master port for reading *//
msgAddr = PUT_GLOBAL_BASE + AccId*NPROCS*PTL_MESSAGE_SIZE +
          msgSrc*PTL_MESSAGE_SIZE;

//*Read the Payload from msgAddr and store it to PayloadBuffer *//
ppmReadAddressSpace(msgPayloadHandle[msgSrc], msgAddr,
                   PTL_MESSAGE_SIZE, PayloadBuffer);
```

```
14
15 for i=0,...,NPROCS: //* Close the master ports *//
16   ppmCloseAddressSpace(msgPayloadHandle[i]);
```

Furthermore, PPM API is used to manipulate the master ports with open, close, read, write operations as described in Code 2. Primarily, NPROCS[8] message Handler are allocated in each accelerator so that each accelerator reads/writes to target accelerator global memory. Subsequently, NPROCS master ports with appropriate names are opened and connect the accelerator(AccId) with all accelerator's local buses. Whenever the accelerator wants to receive[9] one message from certain node, it must identify the source node, message address and temporary buffer for incoming message store. Message address can be computed with global address base as well as the initiator and target identifiers (lines 9-10). At the end of Portals instantiation, all master ports must be closed with **`ppmCloseAddressSpace`** OVP command. With similar way other four master ports have been implemented, while this way can be used for any type Accelerator development using OVPsim.

3.3.2 Slave Ports

In previous subsection portals message exchange was described using master ports. Except from data packet placement in correct positions according to initiator node, accelerator needs a mechanism to be triggered in right time. Hence, a number of slave ports are used so that the accelerator can be respond to bus transactions. Each accelerator is triggered from three events (one for each incoming message type) as illustrated in Figure 3.3. PPM API is responsible for slave port creation and ICM API for connection with bus as described in Code 3.

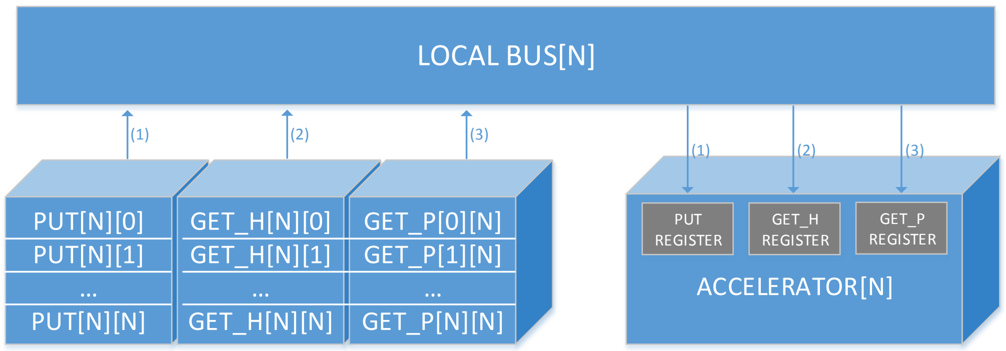


Figure 3.3: Accelerator's slave ports

[8] One handler for each incoming message from Portals Accelerator. Total NPROCS^2 handlers are allocated in our platform.

[9] ppmWriteAddressSpace is used for writing to master port and it has the same parameters with ppmReadAddressSpace.

OVPsim uses especially variables (called registers) as slave port handler (Figure 3.3). Initially, **`PPM_REG_WRITE_CB`** function must initialize these registers, while **`ppmCreateRegister`** creates both a register object **`(put_message_cb_reg`** in Code3) that can be accessed by Imperas Debugger and also reads and writes events that can trigger the Portals Accelerator. Register is accessible through the port associated with the memory region IF_Window. Each Slave Port can transfer information with size REQ_SIZE to peripheral; this property is used to transfer initiator ID so that the accelerator knows the initiator node which sends the trigger request. Subsequently, NPROCS slave ports with appropriate names are opened and connect the accelerator(AccId) with all accelerator's local buses, while the address in each slave port is determined by the range of the bus connection[10]. Moreover, **`PPM_REG_WRITE_CB`** function with the **`put_message_cb_reg`** register is used from Accelerator to handle the incoming portals messages. Finally, slave ports are closed with **`ppmCloseAddressSpace`** OVP command.

Code 3 Create and Connect slave ports to bus

```
1  //* Initialize message CallBacks *//
2  PPM_REG_WRITE_CB(put_message_cb_reg)
3  //* Create NPROCS CallBack Registers to each Accelerator *//
4  for i=0,...,NPROCS:
5    ppmCreateRegister(IF_Window[i], REQ_SIZE, put_message_cb_reg)
6    //* Open NPROCS slave ports to each Accelerator *//
7    ppmOpenSlaveBusPort(PortNames[i], IF_Window[i], REQ_SIZE)
8    //* Connect each slave port to local bus *//
9    icmConnectPSEBus(Acc[AccId], LocalBus[i], PortNames[i],
10   PUT_GLOBAL_BASE + AccId*NPROCS*REQ_SIZE + i*REQ_SIZE,
11   PUT_GLOBAL_BASE + AccId*NPROCS*REQ_SIZE + i*REQ_SIZE+REQ_SIZE-1)
12   //* ...............................................*//
13   PPM_REG_WRITE_CB(put_message_cb_reg){
14     //* Handle Portals message *//
15   }
16   //* ...............................................*//
17 for i=0,...,NPROCS: //* Close the slave ports *//
18   ppmCloseSlaveBusPort(IF_Window[i]);
```

3.4 Portals Accelerator Model

The framework presented in section 3.2 allows integrating under-development hardware within a complete parallel system simulation model, as a result complete software applications can be executed and not just specific, corner case scenarios. Hence, our selection was to initially model our hardware at the untimed functional level[Cai and Gajski 2003] in which the low level timing issues are hided for the sake of simulation execution time, while later on, pure hardware model is developed as described in detail in Chapter 4.

[10] Low & High Address of slave port are determined in line 10 & 11 respectively.

The basic components of Portals accelerator processor are shown in Figure 3.4. Slave ports is used to triggered the Portals Accelerator, while master ports to read/write to global address spaces as described in section 3.3. Functionality of each component is analyzed in the following subsections.

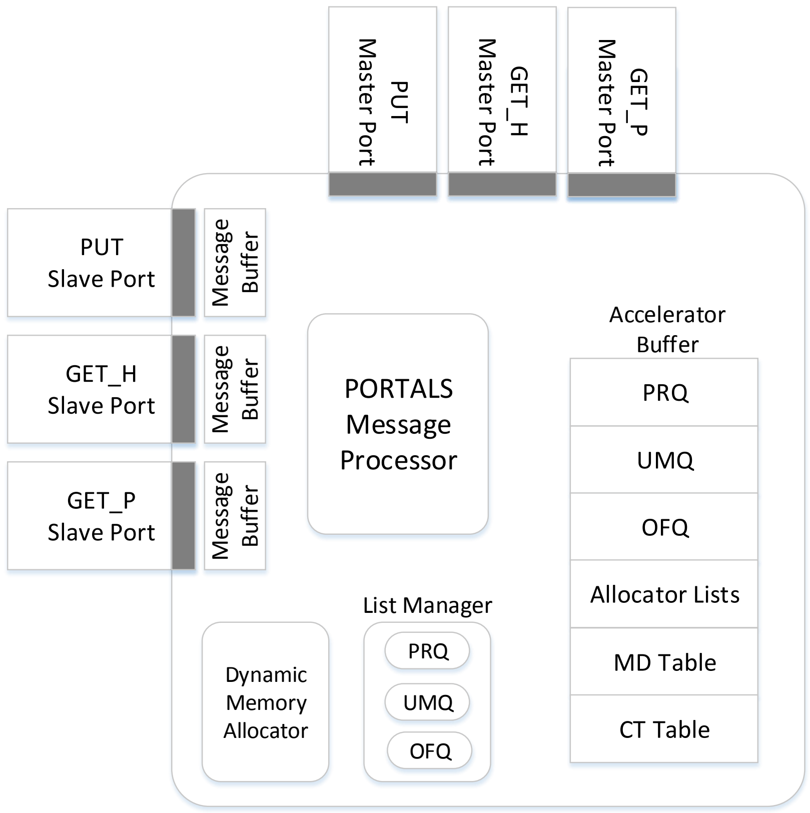


Figure 3.4: Portals Accelerator Processor

3.4.1 Message Buffers

Three Message Buffers are placed at the three slave ports triggering the Accelerator for three different types of incoming Portals messages. These Buffers store the request of incoming messages in order, so that Portals command by processor is transformed to a simple memory write to appropriate address. In this way, the host-processor is instantly allowed to continue its computations just after it writes the Portals command to the Message Buffer. Unlike, master ports do not need Buffers because Header and Payload of Portals messages are stored in Global Memory.

3.4.2 Portals Message Processor

The Portals Message Processor orchestrates the data flow through all Accelerator's components according to the control flow imposed by each Portals command. Initially, when one request message is placed at the message buffer, message processor decodes the 'command type' which determines the type of command and the message (Header and may be Payload) position in Global Address Scheme. Hence, besides the message buffers, the message processor communicates with master ports which have access in Global Address Schemes and with Accelerator buffers. Moreover, the message processor issues certain list operations to the list manager according to message condition and asks for memory space from the dynamic memory allocator so as to save the unexpected messages.

3.4.3 Memory Descriptor & Counting Event Tables

As described in Chapter 2, a memory descriptor contains information about a region of a process memory and optionally points to counting event where information about the operations performed on the memory descriptor are recorded. In other words, whenever initiator issues a Portals command, it must bind a memory descriptor so as to declare the range of User Space memory. As illustrated in Figure 3.5a Memory Descriptor Table is created which contains *MAX_MD_ENTRIES* memory descriptors. Each memory descriptor contains a pointer to UserSpace memory as well as the size of its memory. In addition, ct_handle is used for information about completion of PtlGet transaction, while *used* is a boolean variable for MDTable allocation criteria[11]. Target node ID is used as hash key for allocation criteria so that certain range of MDTable is traversed; for instance, in a system with 64 nodes only *MAX_MD_ENTRIES/64* entries are traversed in each MDBind command.

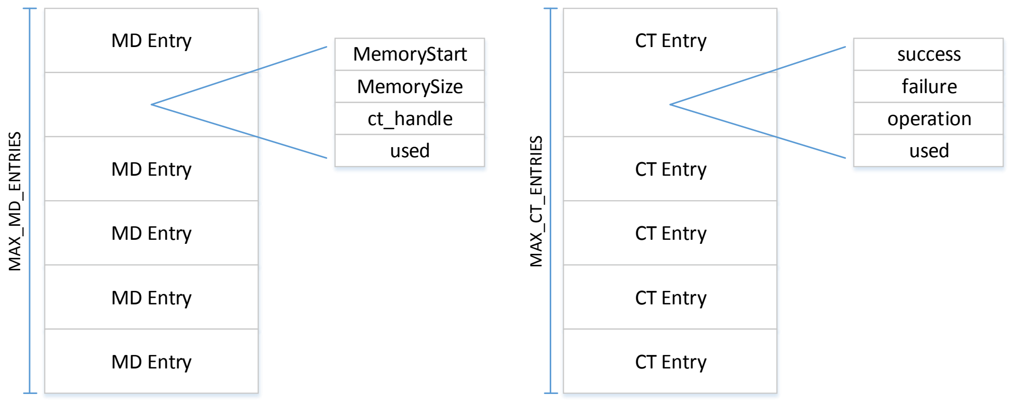


Figure 3.5: (a) MDTable (b) CTTable

Another significant portals structure is counting event which is responsible for completion movements. Each counting event contains both a count of succeeding events and a count of failing events, while operation variable stores the type of Portals movement (PtlPut or PtlGet). Similar to MDTable, CTTable is created which contains MAX_CT_ENTRIES counting events structures as illustrated in Figure 3.5b. With this table, the appropriate event can be searched in constant time since ct_handle is the pointer of CTTable, while PtlCTAlloc command has computation cost (O(MAX_CT_ENTRIES)) in which searches for free entry.

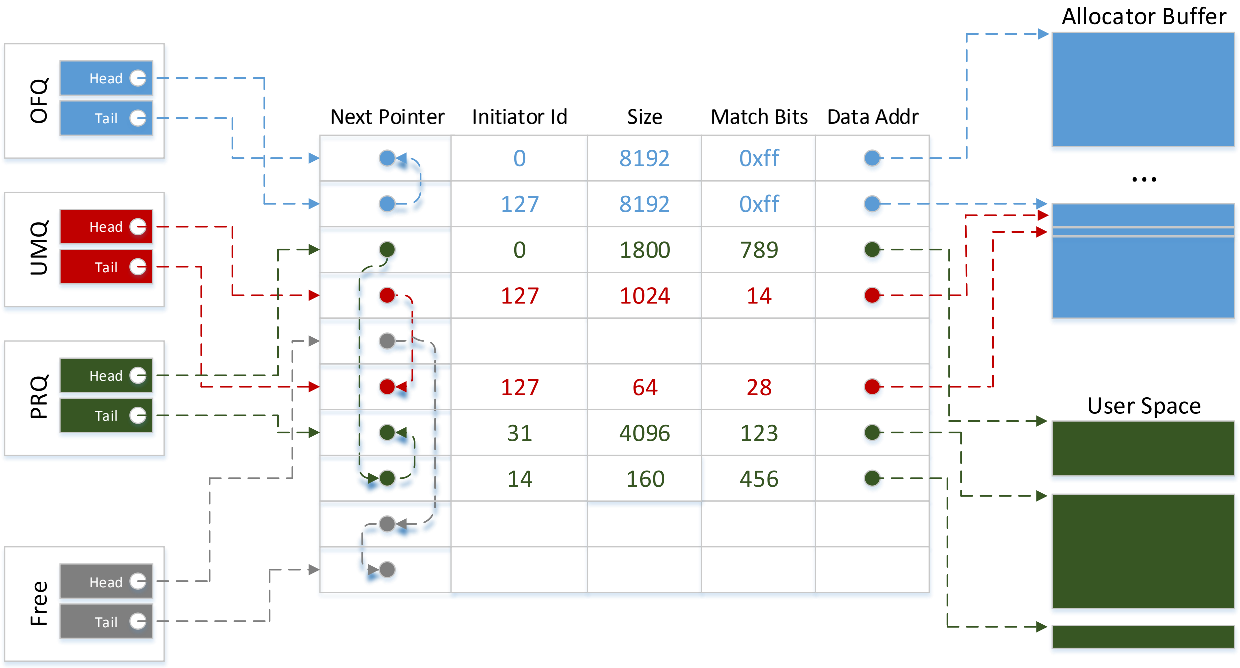
[11] When used is zero the specific entry is free, else this entry is used.

Number of initiator node is used as hash key for computation cost reduction by factor of 64, in 64 nodes platform.

3.4.4 List Manager

The list manager performs three basic list operations, search, insert and delete on the OFQ, UMQ and PRQ lists. In addition, a free list is maintained including all the Accelerator Buffer positions which are not allocated yet by either of the lists. Each list is implemented by a head and a tail pointer. When a new item is inserted in one of the three lists, header of the free list advances to the next free entry, while the newest item is added at the end of the list so as to preserve the message order updating the list (OFQ, UMQ, PRQ) tail's pointer to point to the new element. Similarly, when one item is removed from some of three lists, the tail of the free list is updated to point to that element, while the tail which shows to removed item is updated to point one item back.

Figure 3.6 illustrates the high-level architecture of our list manager in a 128-node platform. OFQ elements are shown in blue, UMQ elements are shown in red, while PRQ and free elements are shown in green and grey respectively. Allocator buffer (illustrated with blue) is accelerator memory which is connected with accelerator through master port



Next Pointer	Initiator Id	Size	Match Bits	Data Addr
●	0	8192	0xff	●
●	127	8192	0xff	●
●	0	1800	789	●
●	127	1024	14	●
●				
●	127	64	28	●
●	31	4096	123	●
●	14	160	456	●
●				
●				

Figure 3.6: Portals List Manager

(Figure 3.2), while UserSpace (illustrated with green) is the application memory. In other words, unexpected message payload is copied in Allocator Buffer, while expected message payload (PRQ) is a simple pointer to user space.

In Portals initiator function, the platform allocates OFQ entries, so that each unexpected incoming message stores its payload. In Figure 3.6, 128 (0-127) OFQ elements with 8192 bytes are allocated; so that unexpected messages with different initiators can be stored in their own allocator buffer. The size of OFQ entry is determined by Portals Init function and can have any size depending on the application, while ignore bits must be 0xff so as to always match according to match function as described in Chapter 2. OFQ entry contains local_offset which shows the allocator buffer section which is used. Initially, local_offset is zero, when unexpected messages arrive the local_offset advances 'message_size' bytes so as to accept the next message payload. For example in Figure 3.6, the payload of first unexpected message is stored to the top of allocator buffer and OFQ local_offset advances 1024 bytes. Subsequently, payload of second unexpected message is stored in 1024 - 1088 allocator buffer position. In case of allocator buffer overflow, implementation of circular array [10] is used as illustrated in Figure 3.7. Whenever payload of unexpected message is greater than (OF_Size minus local_offset), a portion of payload is stored in the bottom of buffer and remainder payload is stored to the top of the buffer. Finally, start variable is used to check the full buffer case.

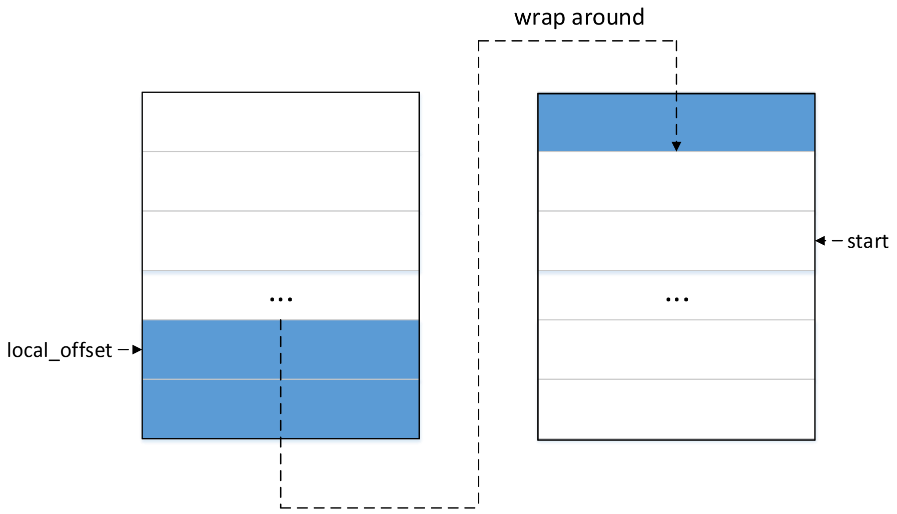


Figure 3.7: Circular Allocator Buffer

As the number of communicating nodes increases, the number of list entries increases triggering an increase in the time needed in order to linearly search them. For this reason, one of the most promising solution to this issue is to utilize a hashing scheme, so that almost constant search times will be enjoyed even at very large systems. When a collision takes place, chaining can be performed by adding the newest item at the end of the list so as to preserve the message order. However, hash-based software implementations are slower, as a number of operations (e.g. calculating the hash function for a key) have to be executed before touching the first entry. For this reason our scheme utilizes a simple hash function which has as hash key the source field of the Portals message. The hashing function assumes that the number of available buckets is a power of two, leading to an efficient hardware implementation. For example, in a hashing scheme with M buckets, a message with initiator id equal to *i*, is assigned to the bucket corresponding to the result of the modulo function *i%M* which for M being a power of 2 reduces to *i&(M-1)*, which is analyzed to a simple binary *'&'* operation and a subtraction by 1.

3.4.5 Dynamic Memory Allocator

As described in the above subsection, during Portals Init function, a number of OFQ entries are inserted with the appropriate Allocator Buffers so as to store the Payload of unexpected messages. The OFQ payload is not critical during OF search and thus it should not consume valuable space in the Accelerator Buffer. So, our Accelerator processor contains a mechanism for allocating and freeing the data corresponding to the payload. Especially, a dynamic memory allocation scheme was implemented in the Accelerator Processor. Our allocator is based on the buddy memory allocation algorithm [11] which can be very efficiently implemented in hardware since it mainly comprises of binary operations.

3.5 Rendezvous Protocol

This section presents the Portals implementation of rendezvous protocol, and how MPI commands can be integrated with rendezvous protocols. As described in Chapter 2, eager protocol ensures asynchronous progress in both expected and unexpected cases. Especially, in expected cases PtlAppend is posted before incoming data and the message is asynchronously delivered in the user buffer, while in unexpected cases PtlAppend is posted after the incoming data begins arriving and either the get request is issued to retrieve the remainder message, or payload is stored in Allocator buffer. However in case of unexpected messages eager protocol wastes the bandwidth in payload retransmission or uses enormous amount of allocator buffer in long messages. In contrast, traditional rendezvous protocols (presented in Chapter 2) ensures asynchronous progress only for unexpected messages, as the header data is immediately available when PtlAppend is posted, while the protocol does not ensure asynchronous progress for expected messages, as the target node must enter the library after the header arrives to issue the get request. In other words the target processor should be interrupted to calculate the PtlGet arguments after the header arrives.

Portals provides a mechanism through which an application can schedule message operations that initiate when a counting event reaches a threshold. Operations which contain this mechanism (using Portals counting events) are called triggered operations.

In this work PtlTriggeredGet operation is implemented by extending the PtlGet argument list to include a counting event on which the operation will trigger and a threshold at which it triggers. So, triggered rendezvous protocol is implemented utilizing Portals Triggered operations to issue the target-side get request without involving the host processor as illustrated in Figure 3.8. The first *eager_limit* bytes of the message are sent to the target when the PtlPut is posted. If the message is expected, the first part of the message is delivered directly into the target UserSpace buffer, otherwise it is delivered into bounce buffers. A counting event which counts bytes delivered is attached to the target buffer, and a triggered get is scheduled to execute when a message larger than the eager limit arrives. The counting event is modified whether the message is expected or unexpected, so the protocol provides asynchronous process in either case.

Figure 3.8 presents the MPI send-receive non-blocking commands and how they manipulate the Portals routine to use triggered rendezvous protocol. Especially, MPI_IRecv and MPI_Isend commands call PtlTriggeredGet, PtlAppend and Ptl_Put operations respectively. In an expected scenario, PtlTriggeredGet operation is called but Get request is posted just the first part of message arrives in target side using PtlCTInc command, otherwise (unexpected scenario) just the first part of the message arrives, a counting event is created and takes the threshold value. Later on, when MPI_IRecv is placed, Get request is sent immediately, as the counting event has the appropriate threshold value. Finally, in both scenarios, matching information for the get movement must be pre-calculated, rather than retrieved from the header data.

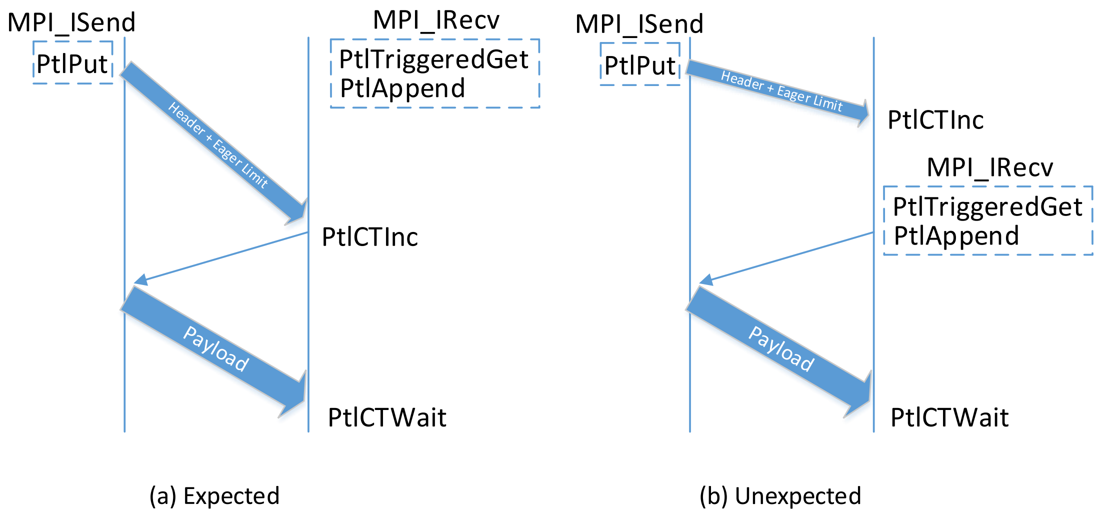


Figure 3.8: Communication pattern for triggered rendezvous protocol

3.6 One-sided Communication

Initially, this section introduces one-sided implementations as presented in the literature emphasizing their drawbacks in the most of them. Subsequently presents our novel one-sided Portals implementation, and finally it presents the Global Arrays implementation using the proposed one-sided implementation.

As presented in the literature, there is an abundance of implementations of one-sided communication. For instance in [12] are presented three different implementations of one-sided communication using two-sided semantics. Unfortunately, all of these implementations are sub-optimal since they use send-receive commands in both initiator and target node, while they are computation-intensive because they use IProbe and blocking Receive operations as illustrated in Figure 3.9. Particularly, Figure 3.9 illustrates One-Sided Get operation using (a) Two-Sided semantics, (b) Two-Sided semantics in a Multi-thread target implementation and (c) Multi-thread target implementation with Shared Memory. In first scenario, target uses one thread for communication and computation, which calls IProbe command constantly wasting the resources of this thread. Second scenario uses two thread in target side, a communicator thread (Pi) and a process thread(Pj). In this case communicator thread is responsible for target communication calling IProbe routine, while the process thread communicates only with Pi whenever it needs the incoming message. Finally, in third scenario shared memory is used for on-node communication to reduce the number of memory copies and eliminate superfluous communication with the communicator thread.

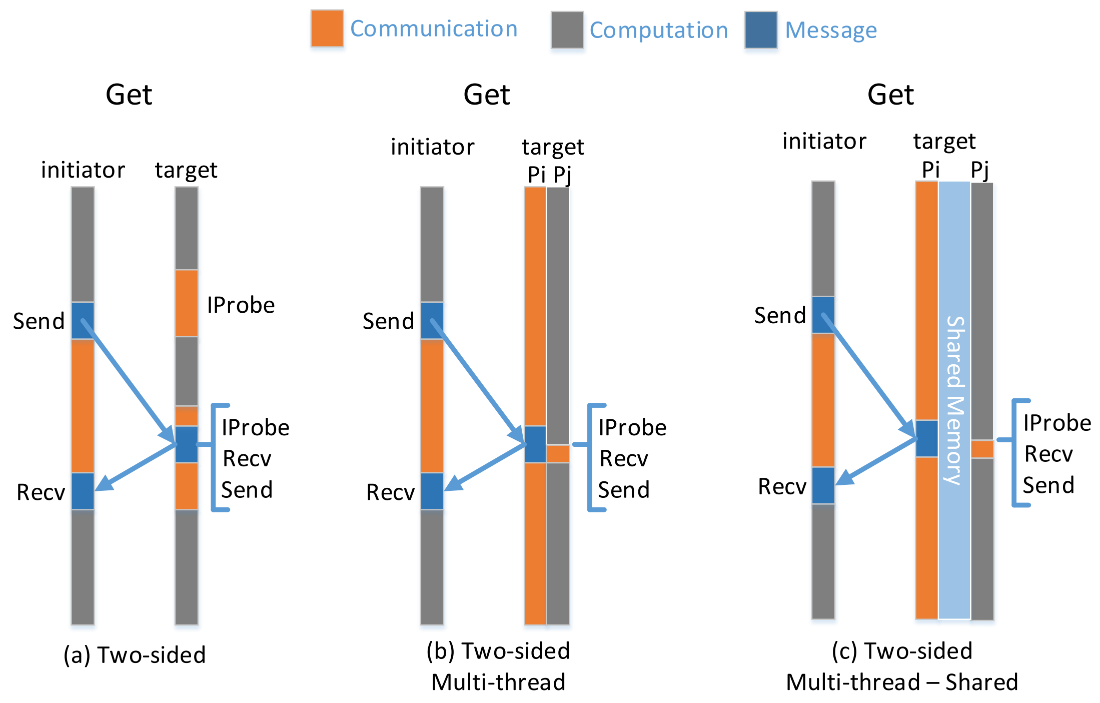


Figure 3.9: One-Sided communication protocol using Two-Sided communication Protocol

3.6.1 Our implementation of One-sided Communication

This subsection presents our novel implementation of one-sided operations eliminating the above common drawbacks. Especially, in our implementation the target of a remote operation is not a virtual address but the ultimate destination in memory of an incoming message is determined at the target node by comparing contents of the message header with the contents of structures at the destination. In more details, target node inserts PRQ entries with unique match_bits[12], which points to accessible UserSpace Buffers as illustrated in Figure 3.10. In Priority List one cluster is added which manages the one-sided entries from all nodes using PTL_SRC_ANY option. Figure 3.10 presents the additional cluster (One-Sided cluster), and the conventional clusters[13] which are used by two-sided communication with our hashing scheme. Hashing scheme is not used in One-Sided communications because target node doesn't know the initiator node during priority entry insertion and the number of Priority One-Sided entries is quite small; as the target usually allocates large amount of UserSpace (as pseudo-shared memory), while the initiator can access on different positions of it through remote_offset.

[12] Each One-Sided Priority entry uses unique match_bit so as to separate the Buffer requests.

[13] In Figure 3.10 illustrates 64-node platform with four buckets, so each cluster accepts messages from 16 different nodes.

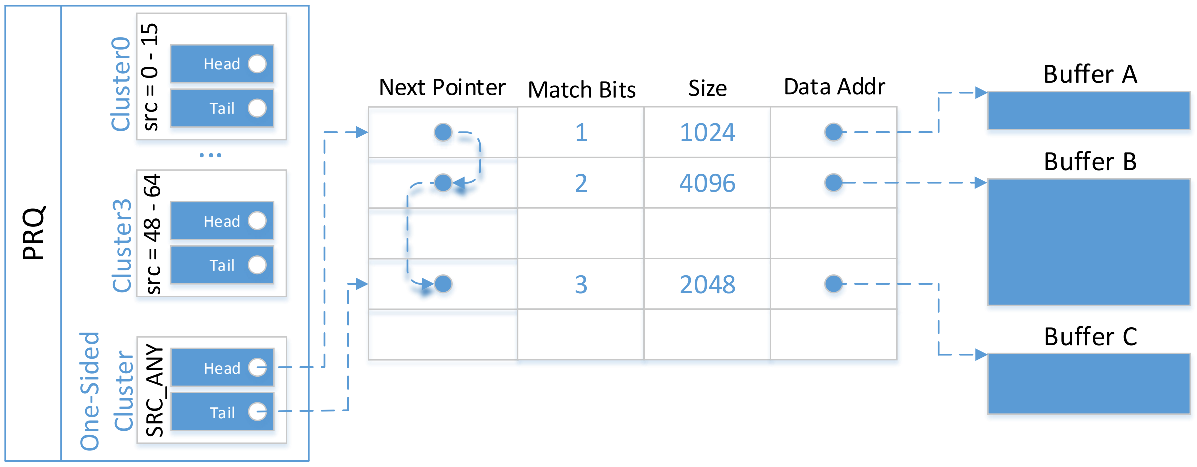


Figure 3.10: PRQ extension to support one-sided communication

Subsequently, such as all commons one-sided communication protocols, the user must allocate memory and then exposes it in a window. For this reason PtlWinCreate command is implemented which inserts One-Sided Priority List entry using a unique match_bits, and subsequently announces the match_bits to all One-Sided participate nodes through Broadcast collective routine as described in Code 4. Moreover, PtlWin structure is implemented which keeps all information about One-Sided participants nodes and their match_bits.

Code 4 PtlWinCreate (void * base_buf , Uns32 size, Ptl_win win)

```
src = nodeId(); //* Get node ID *//
if (base_buf != NULL){ //* Create PRQ entry with base_buf addr *//
  win->match_bits[src] = AssignUniqueMatchBit();
  PtlPRInsert(base_buf, size, win);
}
else{ //* Set the node's match_bits equal to zero *//
  win->match_bits[src] = 0;
} //* Send match_bits to other participant nodes *//
BroadCast(win->match_bits[src]);
```

PtlWin is a simple structure which contains MatchBits and IntraNodes arrays as illustrated in Figure 3.11. IntraNodes array has NPROCS elements with 1 bit size which contains the nodes which participate in One-Sided communication, while MatchBits array saves the match_bits of each participant node. If match bit is equal to zero, this node hasn't memory for sharing, while some node can participate in One-Sided communication and its match bit is equal to zero[14] (as node 1 in Figure 3.11).

[14] In this case the node can only use One-Sided operations to fetch or put remote data, while can't accept One-Sided operations from other nodes because it hasn't pseudo-shared memory.

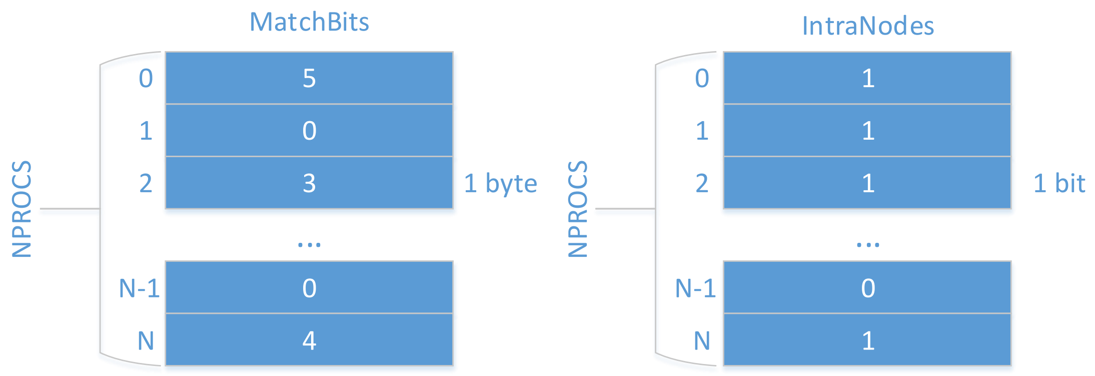


Figure 3.11: PtlWin Structure

Hence, each participating node knows the match_bits of other participants using MatchBits array. For instance, in a 64-node system if node 0 wants to issue a remote put/get operation to node 2, it can find the node 2 match_bit in O(1) from the MatchBits array. Subsequently, when the message arrives in node 2, only the One-Sided cluster is traversed so as to find the appropriate PRQ entry and the appropriate buffer memory concurrently. In all cases, the target must have called PtlWinCreate before incoming One-Sided messages arrives. Hence, One-Sided message is never unexpected since PRQ entry was inserted by PtlWinCreate command. As a result, UMQ and OFQ Lists are never traversed in One-Sided communication, while in case of a PRQ entry is not found, fatal error is occurred.

3.6.2 Global Arrays Implementation through One-Sided Communication

Global Arrays provides a shared memory style programming environment in the context of distributed array data structures (called "global arrays"), while from the user perspective, a global array can be used as if it was stored in shared memory. This subsection presents our Global arrays implementation using One-Sided communication commands.

Initially, one table (called GATable) is created with MAX_GA_ENTRIES[15] of GAEntry structure[16] for Global Array information. GAEntry, in turn, contains the necessary Global Array characteristics, such as its name, type (integer, float, double), number of dimensions (ndim), elements number of each array dimension (dims), pointer of node pseudo-shared memory (buffer), and one Ptlwin entry as illustrated in Figure 3.12.

[15] Implementation can support up to MAX_GA_ENTIES different Global Arrays.

[16] Each entry refers to one Global Array.

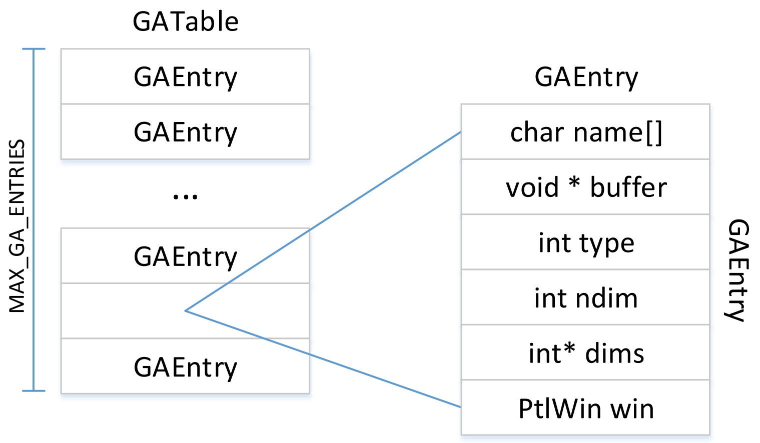


Figure 3.12: GATable implementation

Code 5 shows our implementation of NGA_Create routine. NGA_Create must be called from all participate processors so as to divide the Global array in equal sizes and distribute it in each processor. Initially, it searches for available GATable entry and save the GA characteristics on it. Subsequently, it computes the array size (see next paragraph) which allocate it (with conventional UserSpace allocation scheme (malloc)) from this node. Finally, it creates one window using PtlWinCreate command so as to acknowledge its match_bits to other participants nodes.

Code 5 NGA_Create (int type, int ndim, int ndims[], char array_name[])

```
Crete one Global Array, divide it and distribute it in each processor
Output: Return index of GATable

me = nodeId(); //* Get node ID *//
//* Index of available GATable Entry *//
GAIndex = SearchAvailableGATableEntry();
//* Set the GA characteristics *//
GATable[GAIndex].name = array_name;
GATable[GAIndex].type = type;
GATable[GAIndex].ndim = ndim;
GATable[GAIndex].dims = dims;
//* Compute the array size for each node *//
size_of_thread = SizeOfNode(me,ndim,dims);
GATable[GAIndex].buffer = allocate(size_of_thread); //allocate it
//* Create one window and return the win properties *//
PtlWinCreate(GATable[GAIndex].buffer, size_of_thread,
             &GATable[GAIndex].win);
return GAIndex;
```

Figure 3.13 shows Global array transformation and the pseudo-shared memory division in each participant node. For instance, in Figure 3.13(a) is created a Global array with 9 elements, as a result all nodes must allocate 2 elements except from node0 which must allocate 3 elements. Similar, in Figure 3.13(b) a Global array is created with 11 elements, as a result the first three nodes (0 up to 2) allocate 3 elements while node 3 allocates only 2 elements. In other words, the modulo of Global Array begins to share from nodes with the smallest id to bigger id. This procedure divides the Global Array with fairness; as the maximum divergence among the nodes is 1 element.

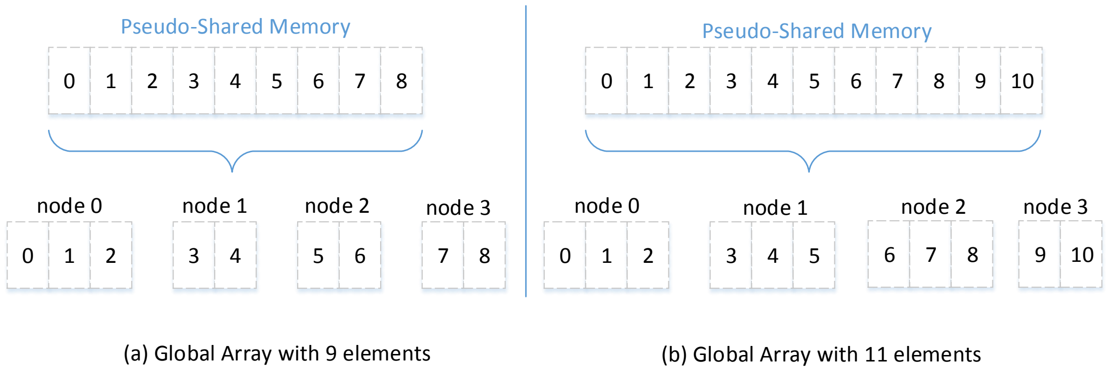


Figure 3.13: Global Array division

Code 6 shows the basic transformation functions for targetID and RemoteOffset computation giving the Global Array Index, as well as the number of GA elements giving the number of node. For example, in Global Array with 9 elements, the 5th Global Array element is stored in node 2 with remote_offset 0, while in Global Array with 11 elements, the 5th element is stored in node 1 with remote_offset 2, etc. Finally, two or greater dimension array is transformed to one-dimension array so as to compute the target and remote_offset in the same way.

Code 6 Global Array Transformation Functions

```
Input:  Index of Global Array element
Output: Node which this element is stored
int Target(int index, int total_size){
  int mod = total_size % NPROCS;
  int div = total_size / NPROCS;
  return (index<= (mod*div)+mod)? index/(div+1): (index-mod)/div;
}

Input:  Index of Global Array element
Output: Offset of node's memory which this element is stored
int RemoteOffset(int index, int total_size){
  int mod = total_size % NPROCS;
  int div = total_size / NPROCS;
  return (index<= (mod*div)+mod)? index%(div+1): (index-mod)%div;
}

Input:  Node Identifier
Output: Number of Global array elements which are stored in this node
int SizeOfNode(int me, int total_size){
  int div  = total_size / NPROCS;
  int flag = (total_size % NPROCS) > me;
```

```
    return div + flag;
}
```

Moreover, all global array functions are implemented minimizing the communication cost. For instance, if one node wants to fill the whole one-dimension array it must call the NGA_Put[17] function only one time as shown below:

NGA_Put(g_a, lo, hi, buf);

In a simple version of NGA_Put, it issues N PtlPut operations to send the data to appropriate node using the above transformation functions. In our implementation NGA_Put groups the PtlPut calls which have the same target ID. For example, in a 4-node platform, the user creates one-dimension global array with 16 elements, so each node contains 4 element of global array. If node 0 calls NGA_Put, then only four PtlPut transactions (with four elements each one) are called, instead of 16 PtlPut transaction with one element each one. Finally, an abundance of GA functions[18] not use any of Portals communication function, because each node simple write in its UserSpace memory without communication cost.

3.7 Summary of Implemented Routines

This section summarizes Portals (Table 3.2), MPI (Table 3.3) and GA (Table 3.4) routines which are implemented in this work.

Portals Routine	Brief Description
PtlPut	Initiates an non-blocking two & one-sided put operation
PtlGet	Initiates a remote two & one-sided read operation
PtlAppend	Creates a single Priority or Overflow match list entry
PtlMDBind	Creates a memory descriptor to be used by the initiator
PtlMDRelease	Releases the internal resources associated with a memory descriptor
PtlTriggeredGet	Initiates a PtlGet operation when a counting event reaches a threshold
PTL_Barrier	Blocks until all processes in the communicator have reached this routine
PtlCTAlloc	Allocates a counting event that counts either movement operations or bytes
PtlCTFree	Releases the resources associated with a counting event
PtlCTWait	Provides blocking semantics to wait for a counting event to reach a given value
PtlCTInc	Provides the ability to increment the success or failure field of a counting event
PtlWinCreate	Creates a window for One-Sided Portals Communication
PtlWinfree	Releases the window for One-Sided Portals Communication
PtlWinFence	Synchronizes all Intra-Node[19] processors

[17] In this case, user must assign lo=0, hi = N, where N is dims[0]. buf contains the data which node wants to save to Global Array.
[18] such as GA_Add, GA_Scale, GA_Zero
[19] Intra-Node processor is one processor which participate in a common window.

PtlPRInsert	Inserts a PR entry to one-Sided Cluster
PtlPRDelete	Deletes the PR entry to one-Sided Cluster

Table 3.2: Implemented Portals Routines

MPI Routine	Brief Description
MPI_Init	Initialize the MPI execution environment through Portals environment
MPI_Finalize	Terminates MPI execution environment
MPI_ISend	Begins a nonblocking send
MPI_Irecv	Begins a nonblocking receive
MPI_Recv	Blocking receive for a message
MPI_Wait	Waits for an MPI request to complete
MPI_Alltoall	Sends data from all to all processes
MPI_Alltoallv	Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data
MPI_Allreduce	Combines values from all processes and distributes the result back to all processes
MPI_Reduce	Reduces values on all processes to a single value
MPI_Bcast	Broadcasts a message from one process to all other processes
MPI_Barrier	Blocks until all processes in the communicator have reached this routine
MPI_Comm_rank	Determines the rank of the calling process in the communicator
MPI_Comm_size	Determines the size of the group associated with a communicator

Table 3.3: Implemented MPI Routines

GA Routine	Brief Description
GA_Init	Initialize the GA execution environment through Portals environment
GA_Terminate	Terminates GA execution environment
GA_Nodeid	Determines the rank of the calling process in the communicator
GA_Nnodes	Determines the size of the group associated with a communicator
NGA_Create	Creates Global Array with pseudo-shared memory
GA_Destroy	Destroys the Global Array from all nodes
GA_Duplicate	Duplicates an existing array
NGA_Put	Performs remote write. The data is simply accessed as if it were in shared memory
NGA_Get	Performs remote read. The data is simply accessed as if it were in shared memory
NGA_Acc	Performs atomic remote update to a patch (a section of the global array)
NGA_Read_inc	Performs atomic remote update to an element in the global array
GA_Print	Prints the entire array to the standard output
GA_Fill	Assigns a single value to all the elements in an array
GA_Sync	Acts as a barrier, which synchronizes all the processes and ensures that all the Global Array operations are complete at the call

GA_Add	Adds two arrays and saves the results to third array
GA_Dgemm	Performs matrix multiplication: C := alpha * A * B + beta * C, where alpha and beta are scalars, and A, B and C are matrices
GA_Zero	Sets all the elements in the array to zero
GA_Scale	Scales all the elements in the array by ***val*** factor
GA_Copy	Copies the contents of one array to another
GA_Dgop	Combines values from all processes and distributes the result back to all processes
NGA_Distribution	Finds out the range of the global array that each process owns
NGA_Access	Provides access to local data in the specified patch of the array owned by the calling node

Table 3.4: Implemented GA Routines

4

From Virtual Accelerator to Silicon

This chapter presents the most significant enhancements/changes needed so as to transform the functionality of the Portals accelerator to silicon as well as micro-architecture exploration for an efficient implementation for our design. Finally, it describes the implementation of multi-node H/W platform and the verification methods which is used so as to verify our cycle-accurate accelerator behavior.

4.1 Introduction to High Level Synthesis

Since the last decade, the electronics industry has been challenged by the increasing complexity of digital systems combined with tight time-to-market schedule. In order for chip design projects to succeed, designers must create and verify differentiated hardware more quickly than their competitors. Unfortunately, today's design flows begin by manually writing an RTL description, modeling the C/C++ function behavior. Moreover, the RTL description codifies the micro-architecture that will be implemented, largely determining the final Performance, Power and Area (PPA) with minimal ability to explore more optimal alternatives. This inflexibility has negatively affected the risk/reward tradeoff of introducing new hardware.

Figure 4.1 illustrates the productivity of new design using different abstraction layers as described in [13]. Starting with the transistor level, moving up to gate level, and then to RTL, hardware design productivity has kept pace with the advances in silicon capacity. In the last decade the productivity has lagged because the shift from RTL to higher level descriptions has taken longer than expected due to the lack of production-worthy high-level synthesis tools. So, in order to make the next productivity leap, high-level synthesis must deliver an automated path for the entire design from Cycle-Accurate high level descriptions to RTL while delivering PPA that is at least as good as what is achieved with handwritten RTL.

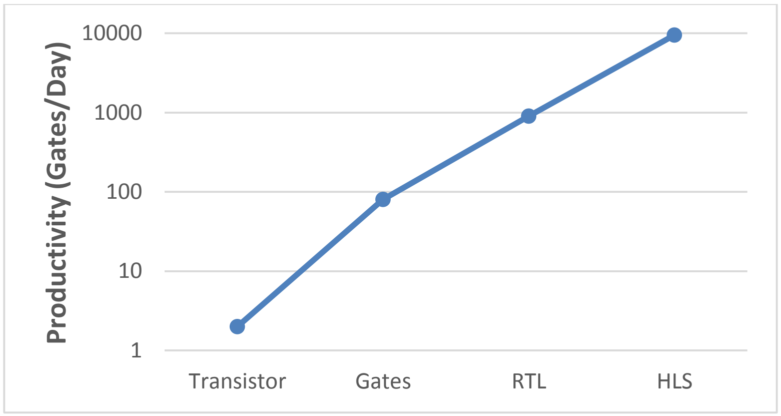
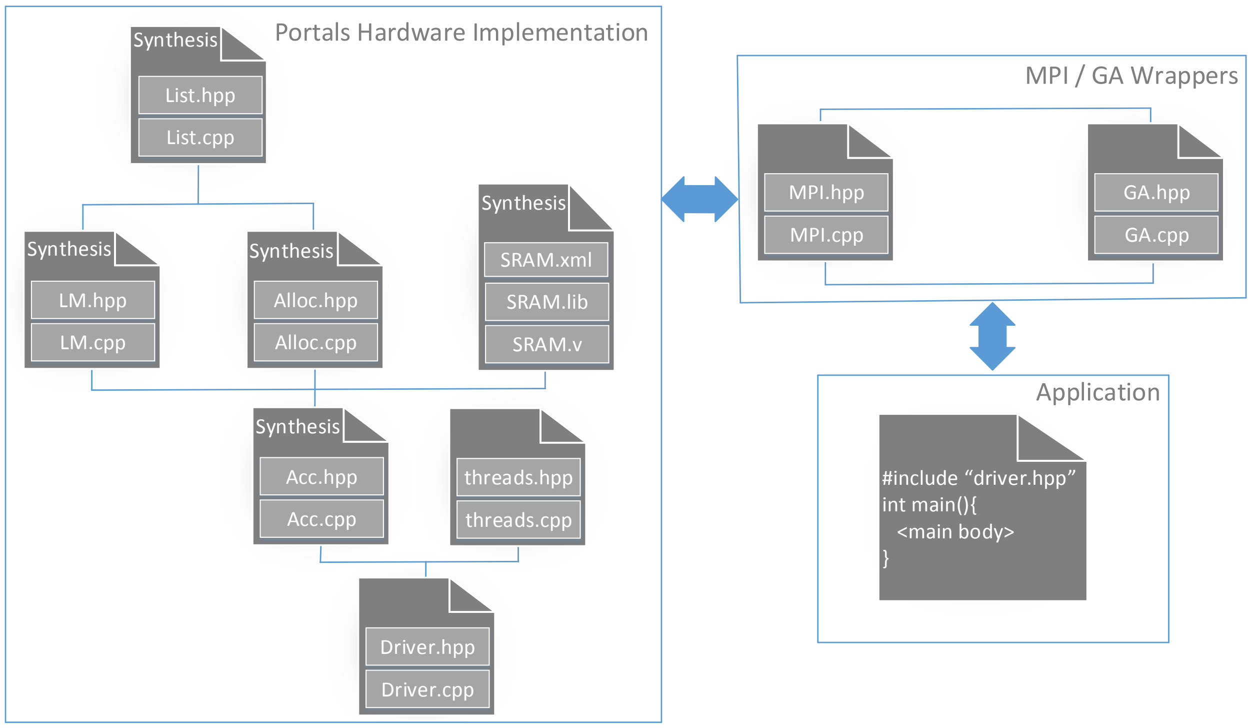


Figure 4.1: Productivity of new design using different abstraction layers

Until now, over 30 industrial and open-source High Level Synthesis tools are released as described in [14]. In this work Cadence C-to-Silicon compiler is selected which reads in a SystemC description of the hardware architecture and generates a Verilog RTL micro-architecture utilizing the high-level constraints that are unique to the target product requirements and process library [13]. The implementation constrains are kept separate from the designs functionality, as a result the same verified SystemC model is easily re-targeted for different end products with different requirements and process libraries. Finally, Ctos contains an amount of verification methodologies at the implementation process; one of them eliminates the most bugs before RTL is even created while if bugs need to be fixed later during implementation the automated engineering change order (ECO) capability is applied so that the project can stay on schedule.

4.2 Hierarchy of Hardware Implementation

This section presents an abstract of our novel Portals Hardware Acceleration implementation. Especially, Figure 4.2 illustrates the SystemC files which are used in this work. Each archive icon consists from its implementation file (.cpp) and a header file(.hpp) so that the below modules in the hierarchy can connect with it.



```
#include “driver.hpp”
int main(){
   <main body>
}
```

Figure 4.2: Hierarchy of H/W SystemC files

Figure 4.2 is separated to three frames, the Portals Hardware implementation frame, MPI/GA wrappers frame and application frame. Initially in first frame, top of the module is a list Hardware implementation with insert/search/delete operations for the three Portals list and allocator implementation. At the next hierarchy layer, there are two basic Accelerator modules; the list manager implementation which orchestrates three fast lists modules (one for each Portals List) and the allocator implementation which is used for dynamic memory allocation for the Unexpected message payloads, while in the same layer, vendor ram is implemented for the UM payload buffer with handwritten RTL Verilog as described in detail in the next sections. Subsequently, functionality of Portals routines are implemented by accelerator module, while a multithread platform is implemented so as to evaluate the effectiveness and accuracy of our Hardware Accelerator. Furthermore, a driver module implements the Portals routines using our novel H/W Accelerator and connects them with our multithread platform as described in detail in the next sections. In second frame, mpi and ga drivers implements the corresponding routines as described in Chapter 3 using Portals implemented routines in H/W. Finally, in third frame is the application which uses either MPI or GA routines. All files are written in SystemC language, while list, listmanager, allocator and accelerator files are written in synthesizable SystemC.

4.3 Transformation to Cycle-Accurate SystemC

This section shows the most significant enhancements/changes needed so as to transform the functionality of the Portals accelerator in C to a fully synthesizable SystemC module. Those transformations can act as a guideline for any accelerator design using the proposed design flow.

4.3.1 Module Interfaces and Synthesizable Code Style

In our high-end simulation environment, the behavior of the hardware is described in separate virtual entities each one comprised of a set of procedures/functions (implementing the actual functionality) and data structures (implementing the underlying hardware structures). For example, our list manager entity is implemented as a set of insert, search and delete functions together with a set of structures modeling the lists and the underlying allocating nodes. Describing such a software entity in hardware requires its interfaces to the other entities and enriching its functionality with timing information.

The following code-segment compares the basic building blocks of our listManager implementation in Untimed C and Synthesizable SystemC respectively. In both implementations SRAMEntry is declared which stores the payload and points to the next element. Untimed C includes special keywords (void*) in order to disassociate the architecture from the data, while hardware description languages (HDLs) have no such capabilities; so C++ templates are used to separate the architecture from the data dependent operations as described in detail in the next section. Furthermore, SystemC SC_MODULE is created to describe our implementation[20]. In addition, in order to convert the untimed high-level software to cycle-accurate hardware implementation, clk and reset signals must be determined as well as all data that feed the entity's functions (sc_in) and those that are the results of the function's execution (sc_out). Moreover, all memories which related with the module are declared in SC_MODULE while the final implementation[21] is determined in scheduling phase as described in the following sections. Subsequently, SC_CTOR constructor describes the type and number of processes which the module can support. For instance our listManager contains four different processes; one for list initialization and three for list operations (insert, search, delete). All processes are implemented with the most common type of SystemC process, the SC_CTHREAD. The main reason behind our selection was that CtoS is able to perform more advanced transforms and optimizations on thread processes than other processes types. For example, CtoS can move operations to different states to

[20] A SystemC module is simply a C++ class that derives from SystemC class sc_module, which can be conveniently specified using the SC_MODULE macro.

[21] CtoS provides several options for mapping an array in SystemC to a physical memory implementation; such as SRAM, DRAM, number of ports, latency etc.

share hardware, insert additional cycles to resolve timing problems, and pipeline loops to improve performance. As a result with this process it is easier to convert untimed C or C++ code to a SystemC thread process.

Untimed C ListManager Header File

```
struct SRAMEntry{
  void *payload;
  void *next;
};


struct ListManager
  List *allocated;
  List unallocated;
  SRAMEntry **SRAM;
};



int searchList( ListManager *list,
  int listId,
  void *element,
  SRAMEntry **previous,
  SRAMEntry **current );
```

SystemC ListManager Header File

```
template <class DataType>
class SRAMEntry{
  DataType payload;
  unsigned long next;
};

SC_MODULE(listManager){
  List allocated[LISTS_NUM];
  List unallocated;
  SRAMEntry SRAM[SRAM_SIZE];

  sc_in<bool> clk;
  sc_in<bool> reset;

  sc_out<bool> itemFound;
  sc_in<unsigned short> listId;
  sc_in<DataType> element;
  sc_out<unsigned long>previous;
  sc_out<unsigned long>current;

  SC_CTOR(listManager):
    SC_CTHREAD(search, clk.pos);
    ...
  }
}
```

Above code-segment is placed in header file so as to declare the interface of the module, while the following code-segment describes the actual module functionality for search operation. Code before the first wait() resets all outputs signals with the write() SystemC function, while the following lines describe the module functionality using an infinite while loop. Since this process describes the intended hardware, the function of the thread should never return, as this would terminate the thread. In contrast it should call wait() to mark the end of a clock cycle and suspend the process until the next clock event. Two significant differences between the Untimed C and SystemC are the iterator and CompareData implementations. Iterator is implemented as a simple unsigned long integer (pointers are prohibited) which points to the next position of SRAM, while CompareData is placed in Message Header in order to disassociate the architecture from the data[22].

[22] Each message Header contains its CompareData inline function, as a result changing Header File CompareData function is changed concurrently.

Untimed C Search Thread

```
searchList( ListManager *list,
  int listId, void *element
  char (*compareData)(void *,void *),
  SRAMEntry **previous,
  SRAMEntry **current
  ){


 SRAMEntry *iter =
      (list->allocated[listId]).head;
 *previous = NULL;
 bool itemFound = 0;

 for(;iter;){
   if(compareData
           (iter->payload,element)){
     itemFound = 1;
     *current = iter;
     break;
   }
   *previous = iter;
   iter=iter->next;

 }
return itemFound;

```

SystemC Search Thread

```
void listManager::search_thread(){
  <Reset all output signals>
  previous.write(0);
  current.write(0);
  itemFound.write(0);
  wait();
  while(1){

   unsigned long iter =
        allocated[listId].head;
   previous.write(0);
   itemFound.write(0);

   for(;iter;){
     if(SRAM[iter].payload.
          CompareData(element)){
       itemFound.write(1);
       current.write(iter);
       break;
     }
     previous.write(iter);
     iter = SRAM[iter].next;
     wait();
   }
  }
}
```

4.3.2 Generic Data Structures

The architecture and the functionality of a hardware module can be independent from data type it supports. Higher level languages like the ones used at the functional level (e.g. C, C++) include special keywords (e.g. void*) in order to disassociate the architecture from the data. Synthesizable portions of the hardware description languages (HDLs) have no such capabilities and hence moving directly from high-level to low-level descriptions requires mixing architecture with data specific code, resulting to error-prone and probably non-reusable code.

In our untimed high-level software an abundance of modules were implemented using void pointers, while in SystemC their functionality were encapsulated in a SystemC module and the architecture was separated from the data dependent operations using C++ templates, as shown in the following code of the list manager's Untimed C and SystemC code respectively.

Untimed C Dynamic Memory Declaration

```
void initFastList(fastList *listP,
  int LISTS_NUM, long SRAM_SIZE
  void (*init_data)(void *)){
  <Allocate 1st dimension of SRAM>
  listP->SRAM = calloc(SRAM_SIZE,
    sizeof(SRAMEntry *));
  <Allocate 2nd dimension of SRAM>
  for(i = 0;i < SRAM_SIZE;i++){
   listP->SRAM[i] = calloc(1,
      sizeof(SRAMEntry));
    init_data(&(listP->SRAM[i])->payload);
  }
  <Allocate Allocated Lists>
  listP->allocated = calloc(LISTS_NUM,
    sizeof(List));
}
```

SystemC Static Memory Declaration

```
template <class DataType, long
  SRAM_SIZE, long LISTS_NUM>

SC_MODULE(listManager){
  ...
  SRAMEntry <DataType>
     SRAM[SRAM_SIZE];
  sc_in<DataType> element;
  ...
  List allocated[LISTS_NUM];
}
```

In the case of Untimed C, the type of Portals queues is assigned dynamically through init_data fynction. In contrast in SystemC the `DataType` is substituted at compile time by a structure modeling a Portals message whereas in the case of memory allocator's list, it is substituted by a corresponding allocation entry. Above code-segment shows the data structure declarations and dependencies in both untimed C and SystemC language.

4.3.3 Cycle Accurate Timing Model

In the case of untimed software, the functions releasing the actual functionality of a software entity are fed with data only whenever they are called. In a hardware implementation, each function is triggered during every clock cycle and thus certain signals should be added in order to mimic the software's control flow. In this thesis, a pair of signals enable_function_name, disable_function_name are added at each function, so that a four-phase hand shake protocol is applied by both the calling and the callee function thread. During the reset phase both interface signals are low. A calling function triggers the enable`_function_name`, while callee function triggers the disable`_function_name`. Whenever the calling function wants to call the callee function, it asserts the enable`_function_name`, while if the callee function is able to process the request it responds by asserting the disable`_function_name`, and begins the execution according to its control flow. When the processing is finished, the callee checks whether the calling function has reset the enable`_function_ name` and if this is the case, it resets the disable`_function_name` signal and returns to its initial state.

Moreover actual hardware requires specifying the functionality that should be processed at every clock cycle. In other words, the operations executed at every state of the control flow should be explicitly defined either source code or micro-architecture specification during scheduling phase as described in the next sections. In case of source code definition the control flow is performed by inserting wait statements which represent the clock registers separating the combinational logic.

The following code-segment shows the four-phase handshake protocol and iteration performing (for MDTable traversing) in MD_Bind function. The wait statement inserted guarantees that each clock cycle performs one for iteration.

Cycle Accurate Timing Model MD_Bind

```
void PortalsAcc::MD_Bind(){
  ...
  while(1){
    while(enable_bind.read() == 0){wait();}
    disable_bind.write(1);
    ...
    <function body>
    for(i=0;i<MD_ENTRIES;i++){
      <processing>
      ...
      MDTable[i].used = 1;
      wait();
    }
    ...
    while(enable_bind.read() == 1){wait();}
    disable_bind.write(0);
  }
}
```

4.4 ListManager implementation

The list manager performs three basic list operations as described in Chapter 3, however as the number of communicating nodes increases, the number of list entries increases triggering an increase in the time needed in order to linearly search them. For this reason, hashing scheme is implemented which assumes that the number of available buckets is a power of two, leading to an efficient hardware implementation. For example the modulo function i%M reduces to i&(M-1). Although in Software implementation this is analyzed to a binary '&' and a subtraction by 1, in hardware no logic is needed, as the hashing function reduces to selecting the M LSBs of i. The above indicate that the hardware implementation of the proposed hashing scheme has negligible overhead. Following code-segment shows our PriorityList hashing function. PriorityList is responsible for both one-sided and two-sided communication. Especially, if the incoming message is two-sided the above modulo function is used, whether if it is one-sided the last[23] cluster is assigned to PRListId. For instance, for PR_LIST_NUM equal to 16, 17 clusters are implemented in total; two-sided communication uses cluster with range 0-15, while one-sided uses the 16th cluster[24].

[23] One more cluster is implemented for OneSided communication in PriorityList.

[24] OneSided communication not uses our hashing scheme as described in the previous Chapter.

Priority List Hashing Function

```
if(OneSidedCommunication == 0){
  PRListId.write(Source & (PRLIST_NUM-1));
}
else{
  PRListId.write(PRLIST_NUM);
}
```

Finally, in this work a Lighter Edition List of the manager is implemented targeting area and power savings. This version uses only one list for PR,UM,OF Lists representation. Hence, different message Header is used for insert, search and delete functions which contain Portals List kind information.

4.5 Micro-architecture Exploration

In the Specifying Micro-architecture step of the CtoS flow, the designer provides additional information to help CtoS implement his design as closely as possible to his design goals[15]. This section presents our micro-architecture declarations for an efficient implementation for our design, focusing in un-resolving loops and memories.

4.5.1 Resolving loops

At many times, in HLS languages there are combinational loops which must be eliminated before synthesis because they cannot be implemented in hardware. A loop is said to be combinational if at least one program execution from the top to the bottom of the loop does not include waiting for a clock edge. In Ctos there are two common operations to resolve the combinational loops, (i) unroll and (ii) break them with different advantages and drawbacks in each one. Completely loop unrolling essentially eliminates the loop increasing the number of operations, because operations inside the loop are copied many times. In other words all loop operations are executed in one cycle increasing dramatically both area and performance. On the other hand, loop braking is another option which simply inserts states in the loop increasing the latency required to implement the behavior and decreasing the area. In this work both of the two above operations are used in different loops. Specifically, if the loop contains a small number of operations inside the loop and the total number of iterations is quite small, then the unroll operation is used, while in case of loops with many iterations break operation is used so as to keep the critical path short. The following code-segment is an example of `higher_power_of_two` function which computes the higher power of two at given 32bit `number_input` and `CT_Alloc` function for `CTTable` allocation. In the first loop, unroll operation is used as the `max_number` of iteration is 32 and the loop contains only one comparator and shift left operation. On the other hand, in the second loop, break operation is used through wait statement as the loop needs to read and write the one port CTTable array[25].

[25] One port Array needs one clock cycle to read or write their elements.

Higher Power of two Returns the power of two which is greater or equal to input number

```
leadZeroPos = 32;
mask0 = 1 << (leadZeroPos-1);
for(;leadZeroPos>=0;leadZeroPos--){
  if(number_input > mask0){
    break;
  }
  mask0 >>= 1;
}
```

Portals CT Table Allocation

```
for(j=0;j<CT_ENTRIES;j++){
  <search for free CTTable entry>
  CTTable[i].used = 1;
  wait();
}
```

4.5.2 Memory Implementation

This subsection presents the possible options for mapping an array to physical memory using Ctos and subsequently it describes the options which are used in this work. Especially, CtoS provides four options for mapping an array in SystemC to a physical memory implementation: (i) flatten, (ii) built-in, (iii) prototype and (iv) vendor array; each option is optimal for a different array use case or access pattern. Flatten option creates an array of registers; this option is best for very small arrays while it increases the number of registers dramatically. Buit-in operation implements the arrays as built-in SRAMs; this is the desired choice when the number of array words is medium (nearly 256 words or fewer [15]), but multi-process access is required. Prototype operation has fastest runtime but it does not represents actual hardware, hence it must be replaced before final implementation. Finally, in vendor option the designer inserts handwritten Verilog RAM implementation; typically a better option for very large arrays.

In this work only built-in and vendor options are used because the flatten option increases the number of registers dramatically, while prototype option does not represent actual hardware. More specifically, the Accelerator Buffer (Figure 3.4) is implemented with fast (2-ports read / 2 ports write) SRAMs[26] using built-in option because these entries should be performed efficiently without having to access the local host memory. Additionally, the buffer for unexpected messages payload is implemented using a 4MB Vendor (2-ports read / 2 ports write) DRAM as described in the following code-segment using handwritten Verilog

[26] 2ports read SRAM can read and fetch two elements per clock cycle, and simultaneously (2ports write) can write two elements per clock cycle.

Vendor DRAM. Moreover, Vendor DRAM technology library is used so as to describe the DRAM constrains while xml file is used to include the above DRAM to whole design.

Handwritten Verilog Vendor DRAM

```
reg [DATA_WIDTH-1:0] MEM [0:NUM_WORDS-1];

//write behavior port
always @(posedge CLK) begin
  if(CE0 & WE0) begin
    MEM[A0] <= D0;
  end
end

 ...

//read behavior port
always @(posedge CLK) begin
  if(CE2) begin
    Q2 <= MEM[A2];
  end
end
```

4.6 Implementation of multi-node Platform

This section presents our implementation of multi-node Hardware platform so that architecture parameters can be evaluated with realistic software programs and not with corner testcases and synthetic benchmarks.

The following code-segment shows the union of SystemC hardware threads to software threads in top module. Initially, top module creates NPROCS SystemC processes with different names and thread numbers (`thread_no`) using SC_CTHREAD. Subsequently, each SystemC thread creates a S/W thread with the given `thread_no`. Hence, main function is created with NPROCS SystemC threads, while `pthread_self` function is used for node identification.

Union of Software - Hardware threads

```
THREADS *node[NPROCS];
<Create Hardware Threads>
for(i = 0; i<NPROCS;i++){
  sprintf(name,"NODE%d",i);
  node[i] = new THREADS(name);
  node[i]->clk (clock);
  node[i]->reset (reset);
  node[i]->thread_no(i);
}
```

```

void THREADS::create_thread(){
  <Each Hardware Thread calls one Software Thread>
  pthread_create(&(tid[thread_no]), &main);
  pthread_join(tid[thread_no]);
}
```

An overview of Hardware Accelerator intercommunication which is used in this work is shown in Figure 4.3. Similar to our Software implementation, one hardware accelerator has 2 local ports (one input & one output) to communicate with its processor, and 2 connection ports to communicate with the other accelerators. Especially, each processor is a Software thread which is created from `pthread_create` function, so it has not inputs and outputs ports[27]; in Figure 4.3 processor is shown to has ports for the sake of uniformity. Moreover, unlike the Software implementation, in hardware implementation each accelerator has only one remote input and remote output[28], as a result it can receive only one message at any time. Hence, in case of two initiators send one message to same destinator at the same time, the destinator can't receive one of two messages. For this reason, one queue is placed in each remote accelerator input with max_size equal to NPROCS[29] as illustrated in Figure 4.3.

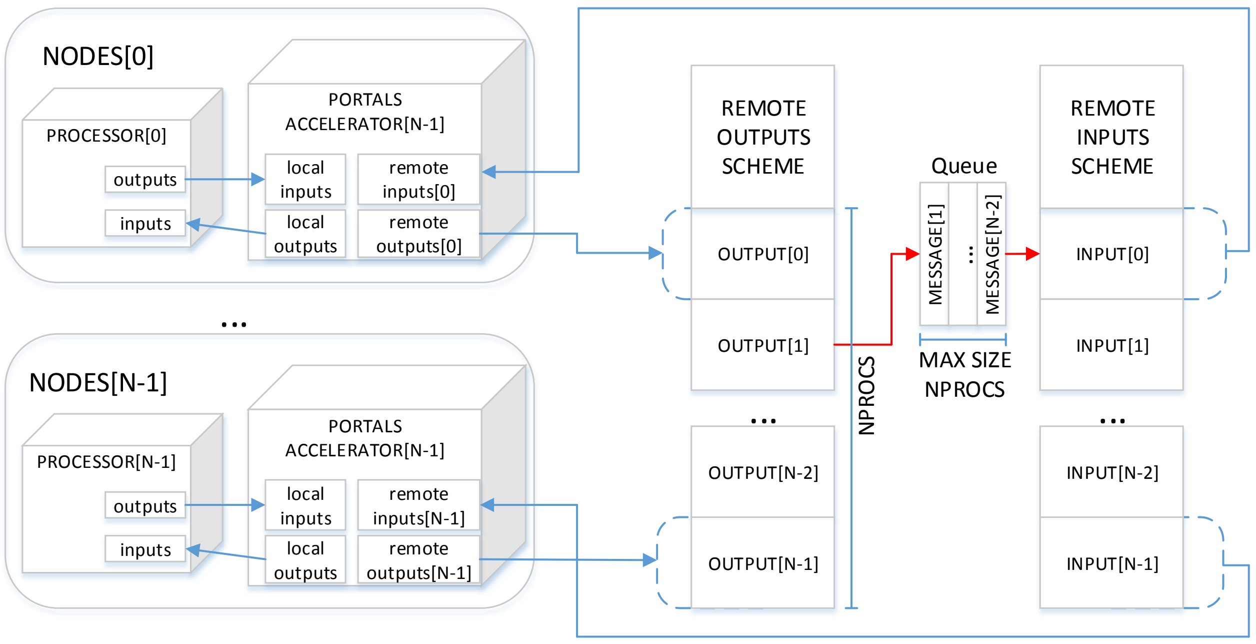


Figure 4.3: Hardware Accelerator intercommunication

[27] Each processor calls its Portals accelerator simply.
[28] In Software implementation each accelerator has NPROCS ports (one for each incoming message from different initiator). In Hardware, this approach is prohibitive because the number of inputs and outputs (and silicon area) are increased dramatically.
[29] This size supports the case of all nodes send one message to same destinator node.

4.7 Hardware Acceleration Verification

A major benefit of SystemC-based high-level-synthesis (HLS) design that is rarely explored is improved verification turnaround and productivity. In other words, a SystemC block design can be expressed in 80% fewer lines of code than RTL, which minimizes the number of potential bugs while promoting functional verification at the interface, function, and protocol levels. Figure 4.4 presents the positive productivity gains in this level of code compaction as described in [16].

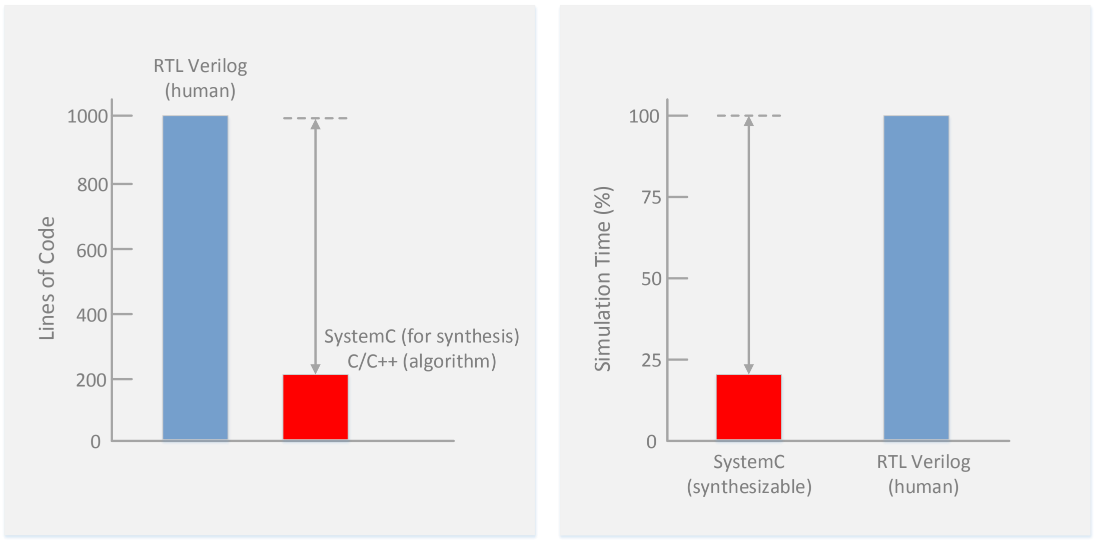


Figure 4.4: Productivity gains with software-driven SoC design

A highly productivity relies on the design and verification progressing concurrently, from a high level of abstraction all the way to gate-level implementation as shown in Figure 4.5 [16]. For verification, the focus is to verify functionality at the highest possible level of abstraction available, and then avoid duplication of effort by directing additional verification activities towards the new and modified design functionality added at each stage of the design refinement process.

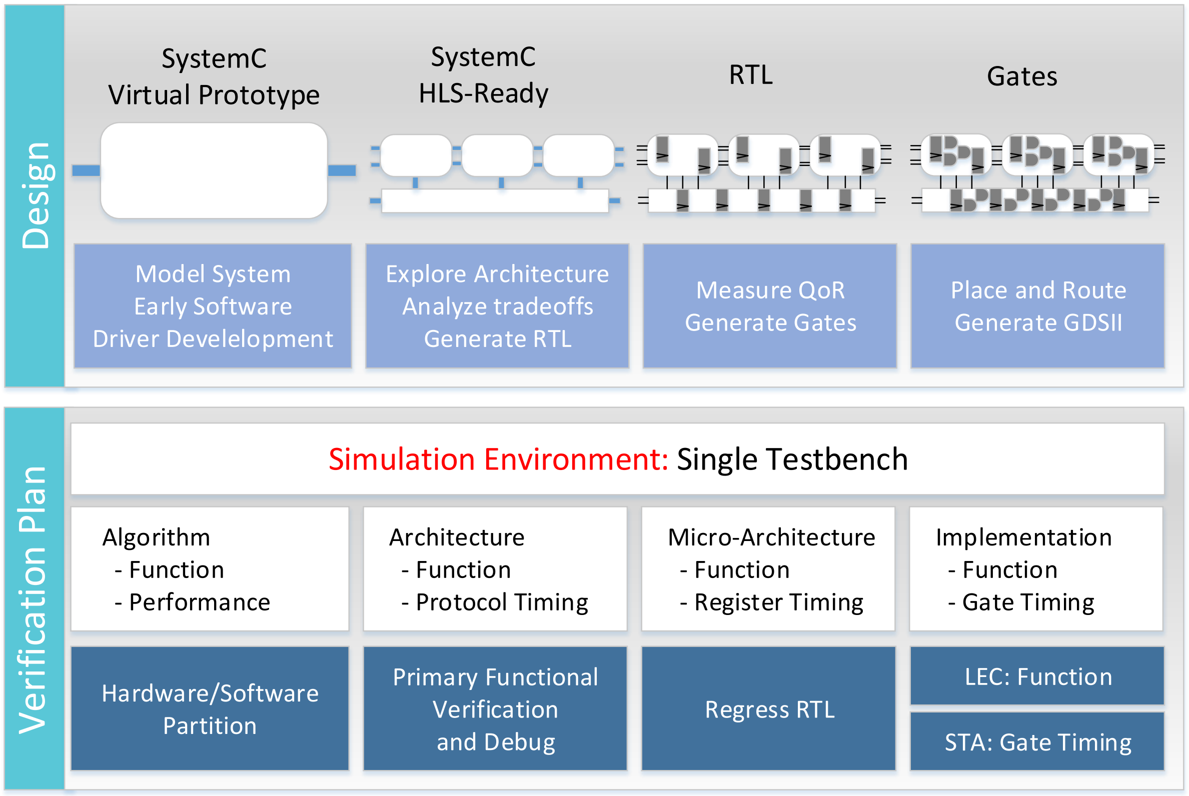


Figure 4.5: Design Flow

The goal is a single common verification environment that spans the different abstraction levels of the SoC design. A single common verification plan is defined such that it outlines the features to be verified in each specific level of design. As architectural decisions are made that bring the design closer to implementation, the verification environment is concurrently extended to test those architectural choices.

Totally, our design is verified through three stages as illustrated in Figure 4.6. Initially, pure SystemC simulation model is verified (Figure 4.6a) so as to compare the SystemC instruction-accurate accelerator behavior with the software one (implemented in untimed C) using wide range of testbenches as described in Chapter 5. In the next step (Figure 4.6b), Verilog behavioral model is generated by Ctos environment with the appropriate wrapper (Top Wrapper), while verification wrapper is used to compare the Verilog behavioral model with the pure SystemC model. Finally in the third verification step (Figure 4.6c) RTL Verilog with exported RAMs is generated by Ctos during the scheduling step using the micro-architecture specifications. Similar to second step, wrapper is used to compare the RTL Verilog model with the pure SystemC model.

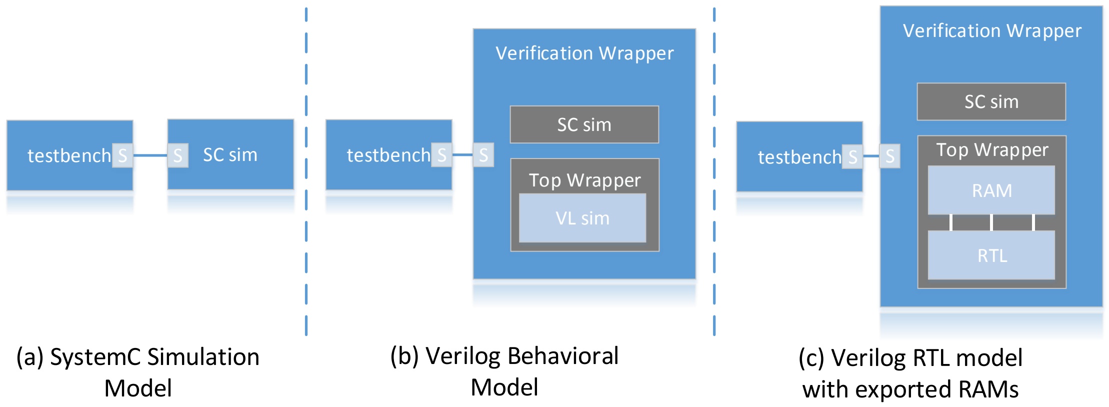


Figure 4.6: Hardware Accelerator Verification Phases

A verification wrapper is a SystemC model with a parameterized constructor that configures the design to be simulated. In addition to the `model under test`, the original SystemC model can also be instantiated as a `reference model` inside a wrapper for `cycle-by-cycle` comparisons of the model under test output and the original SystemC model output as described in the following code-segment for our Portals Accelerator.

Union of Software - Hardware threads

```
SC_MODULE(wrapper){
  ...
  PortalsAcc ref;       // SystemC reference model
  PortalsAcc_ctos rtl; // RTL model
  ...
}

void compare_outputs(){
  int out_ref;
  while(1) {
    wait();
    while(!OUT1_VALID_ref.read()) wait();
    out_ref = OUT1_ref.read();
    while(!OUT1_VALID.read()) wait();
    if( out_ref != OUT1.read() )
      cout << " Verification wrapper has found a mismatch! "
    }
}
```

5

Results

This chapter provides the experimental results of our work. Section 5.1 describes certain benchmarks of widely used NPB suite, and one Molecular Dynamic benchmark in order to prove the accuracy of the introduced framework. Section 5.2 illustrates the abstraction level specification results using HLS tool. Section 5.3 presents the average and maximum search depth of Portals Queues using realistic scenario benchmarks. Sections 5.4 - 5.5 present MPI and GA Results of a wide variety of Routines, and finally, Section 5.6 presents the Area and Power Results of Portals Accelerator. We compare the performance of our novel system with that of a high-end Intel CPU E8400 as well as with an ARM Cortex A9 [ARM 2013] state-of-the-art embedded processor executing the exact same MPI (from the most widely used openMPI library) and GA tasks on top of the Fedora 14 Linux OS and Ubuntu 12.04 Linux OS respectively. The high end processor performance is measured using the Intel VTune [vtu 2012] profiler. The embedded processor is evaluated using the API provided by the OVP simulation for measuring exactly the number of cycles consumed with the average CPI and frequency of ARM A9. Finally, the performance gain of our Portals hardware approach is evaluated with cycle accurate model of SystemC when compared with the same Portals Routines in Software.

5.1 Benchmarks

Four benchmarks of widely used NAS Parallel Benchmarks (Table 5.1) and one molecular dynamic benchmark were used to evaluate the suggested Portals implementation presented in Chapters 3 and 4.

The NAS Parallel Benchmarks (NPB) are comprised of a set of parallel algorithms designed to evaluate the performance of parallel supercomputers [17]. The benchmarks are derived from computational fluid dynamics (CFD) applications, adaptive mesh, parallel I/O, multi-zone applications, computational grids etc. We select IS, FT, EP and DT NAS benchmarks as they utilize a wide range of MPI routines. EP is an "embarrassingly parallel" kernel, which evaluates an integral by means of pseudorandom trials. FT is a 3-D partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long distance communication performance. IS implements a large parallel integer sort algorithm. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance. Finally, DT (Data Traffic) works with randomly generated data using trees and a shuffle as data flow patterns.

Molecular Dynamics (MD) is widely used to simulate many particle systems ranging from solids, liquids, gases, and biomolecules on Earth, to the motion of stars and galaxies in the Universe. We select Molecular Dynamics of Lennard-Jones System benchmark which computes energy fluctuation (using Global Arrays) per particle in a wide range of pressure and temperature values[18].

NAS Parallel Benchmark	Brief Description
IS	Implements a parallel integer sort algorithm
FT	Finds a 3D partial differential equation solution using FFTs
EP	Evaluates an integral by means of pseudorandom trials
DT	Generates randomly data using trees

Table 5.1: NAS Benchmark Suite

5.2 Abstraction level specification results

This section describes the results of the abstraction levels which are used until the cycle accurate RTL Verilog. As described in Figure 5.1 the lines of code is tripled from UntimedC to synthesizable SystemC. The main reason is that synthesizable languages require micro-architecture specification as well as interface definition enriching the functionality with timing information. Especially, in the case of untimed software, the functions releasing the actual functionality of a software entity are fed with data only whenever they are called. In a hardware implementation, each function is triggered during every clock cycle and thus certain signals should be added in order to mimic the software's control flow. Moreover actual hardware requires specifying the functionality that should be processed at every clock cycle inserting wait statements which represent the clock registers separating the combinational logic. Finally, higher level languages like UntimedC include special keywords (e.g. void) in order to deassociate the architecture from the data, while HDLs have no such capabilities and hence the architecture was separated from the data dependent operations using C++ templates.

The goal of HLS is to let hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and through the nature of allowing the designer to describe the design at a higher level of tools while the tool does the RTL implementation. A major benefit of moving to this level of design that is rarely explored is improved verification turnaround and productivity. Our SystemC block design can be expressed in 90% fewer lines of code than Verilog RTL as illustrated in Figure 5.1.

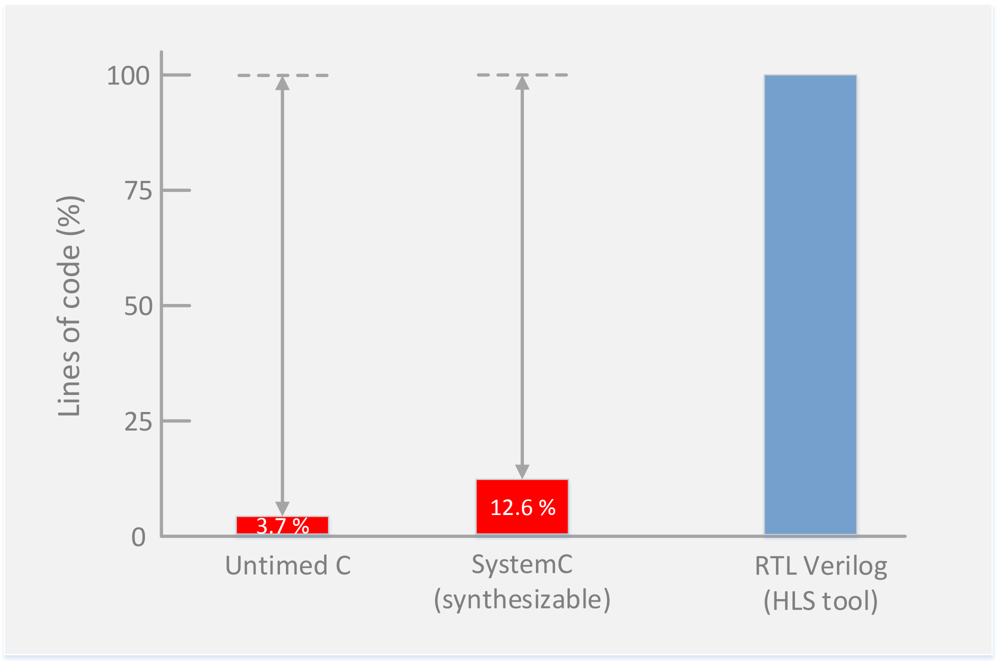


Figure 5.1: Productivity gains in our design using HLS tool

5.3 Portals Queues

This section examines the scalability of the Portal implementation presented in Chapter 3 using the MPI collective routine MPI AlltoAll. MPI_Alltoall is a collective operation in which all processes send the same amount of data to each other, and receive the same amount of data from each other. Hence, it is expected that the size of the Portals queues grows linearly with the number of nodes in the parallel system. Initially, we measure the average and the maximum search depth of Portals queues, observing that as the number of nodes increases the average search depth increases by the same factor, doing the search cost of multi-thousand node platform prohibitive. We attach this problem utilizing our hashing scheme presented in Chapter 3. The results indicate that our hashing scheme with small number of buckets can be traverse the Portals Lists in almost constant time. Finally, we examine a realistic scenario where a host node has received numerous unexpected messages.

5.3.1 Average Search Depth in Portals Queues

This subsection shows the average search depth of Portals queues as measured in our embedded platform as illustrated in Figure 5.2. For the three lists average search depth were shown to grow linearly (x-axis is exponential) with the number of processing nodes. In MPI AlltoAll Routine as described in 5.2 all nodes "do the same work", as a result PRQ and UMQ Lists have the same average search depth (call same number of MPI_ISend and MPI_IRecv Routines). On the other hand the OFQ has greater average search depth from PRQ/UMQ because Portals allocates constant number of OFQ entries during initialization. More specifically, Ptl_Init allocates N OFQ entries (one of each node), where N is the number of nodes. In MPI AlltoAll Routine on average half of messages are unexpected, as a result OFQ has average search depth N/2 approximately[30].

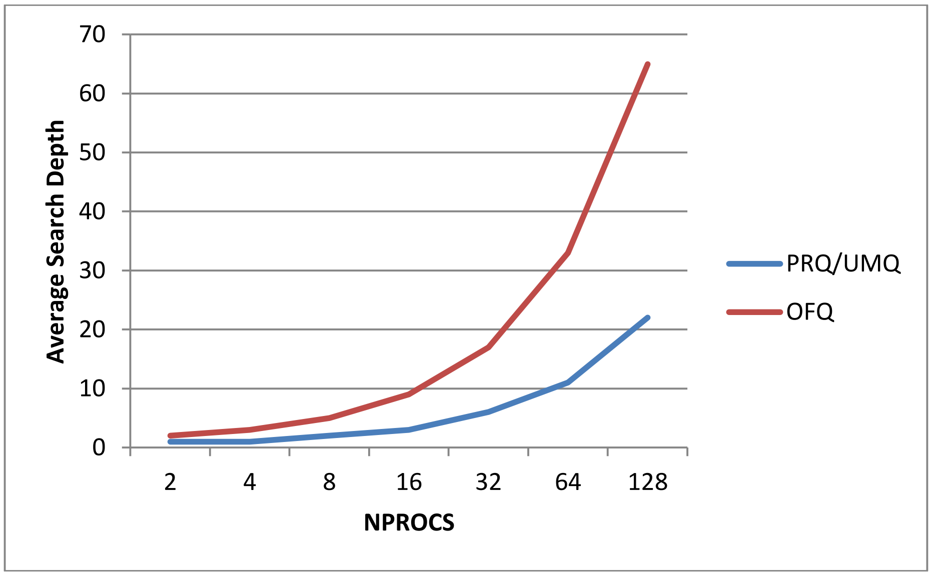


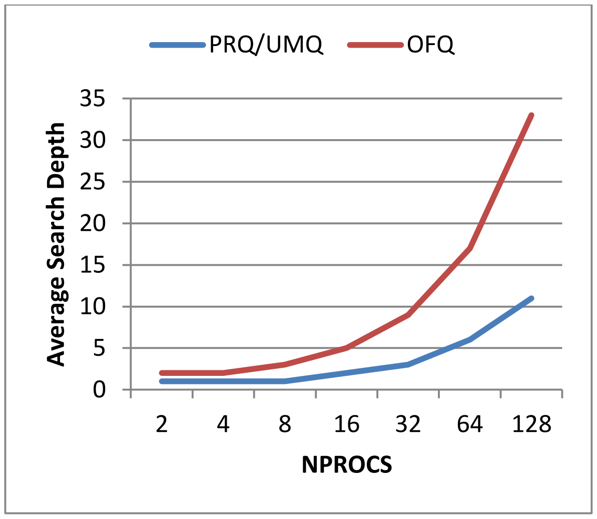
Figure 5.2: Average Search Depth

5.3.2 Average Search Depth in Portals Queues using our Hashing Scheme

As the number of communicating nodes increases, the number of list entries increases (Figure 5.2) triggering an increase in the time needed in order to linearly search them. For this reason we utilize a hashing scheme, so that the lists can be traversed in almost constant search times. Figure 5.3 illustrates the average search depth using our hashing scheme. While

[30] N is the number of nodes.

the curves of Figures 5.2 and 5.3 are identical, the average search depth is decreased by half, doubling the number of buckets.



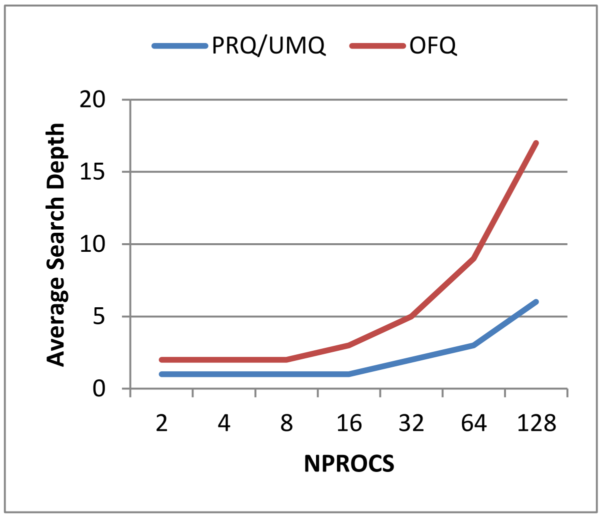
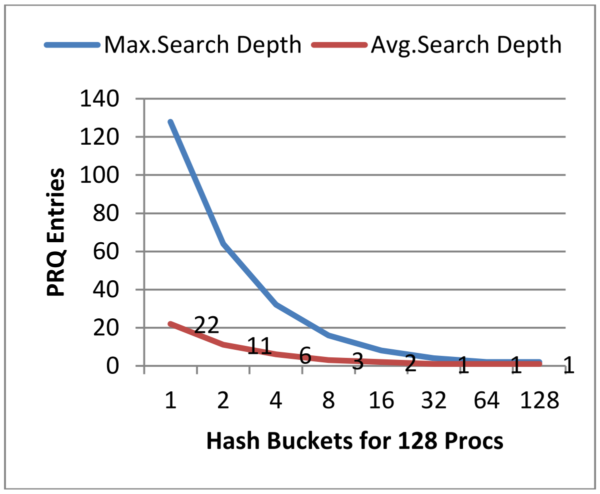
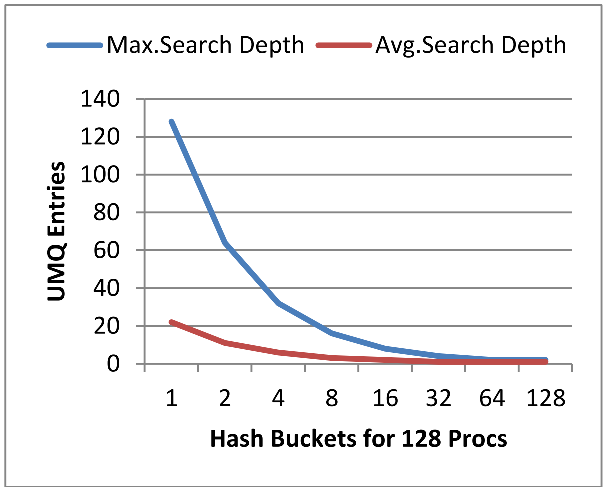


Figure 5.3: Hashing Scheme with (a) two buckets (b) four buckets





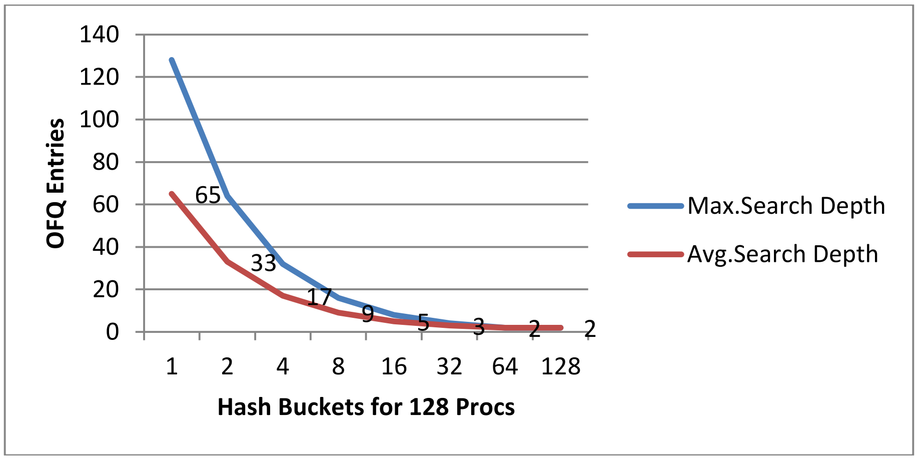


Figure 5.4: Maximum and Average Search Depth for 128 nodes w.r.t. Number of Buckets

Subsequently, Figure 5.4 shows the results of the maximum search depth and the average search depth of the PRQ - UMQ - OFQ when the MPI AlltoAll benchmark is executed. In this experiment, 128 nodes with 1, 2, 4, 8, 16, 32, 64 and 128 buckets were simulated. The configuration with a single bucket matches the one of the conventional serialized list-based scheme. The serialized list-based solution has worst-case depth of 128 entries in all of the three lists. Increasing the number of buckets by a factor of 8 results to decreasing the maximum search depth by the same factor. The above shows that the messages are evenly split to hash buckets. However, the maximum search depth only affects the resources utilized and the average search depth is a better metric for the performance of the system. The same figure demonstrates that average search depth for PRQ/UMQ is 22 entries, which can be reduced to 11, 6, 3, 2, 1 for configurations with 2, 4, 8, 16, 32 buckets respectively, while the OFQ begins with 65 in serialized version, a number which is reduced to 33, 17, 9, 5, 3 with 2, 4, 8, 16, 32 buckets respectively.

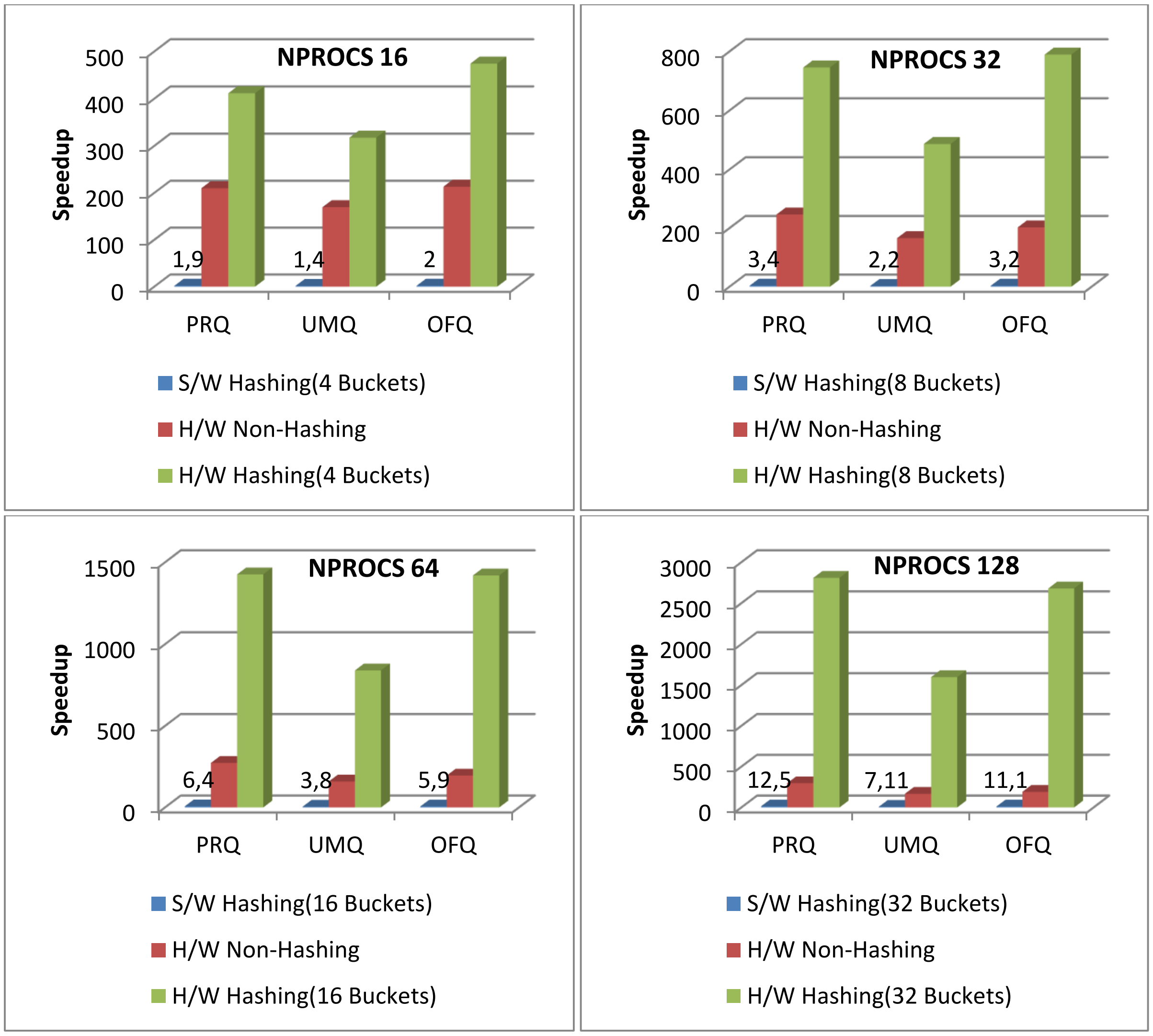


Figure 5.5: Queue Processing Speedup (a) 16 nodes, (b) 32 nodes, (c) 64 nodes, (d)128 nodes

Finally, Figure 5.5 shows the queue processing speedup using MPI Alltoall Routine as a benchmark for 16, 32, 64, 128 nodes and nodes/4 as number of buckets. As the number of nodes is increased by a factor of 2, the speedup is increased approximately by the same factor for Hashing schemes (buckets is increased too), while non Hashing schemes speedup is constant.

5.3.3 Realistic scenario which unloads the UMQ

However, as the number of nodes increases, it is more likely that a certain host node will traverse more and more queue entries before finding a match in the queue. In this section we initially, introduce a certain benchmark which unloads a queue with a predefined number of entries, mimicking the realistic scenario where a host node has received numerous unexpected messages and the matching entry is always found at the tail of the queue. In this benchmark all nodes send unexpected messages to node 0 with tags in descending order, while the node 0 receives messages with tags in ascending order. For example, Figure 5.6 shows the UM List of node 0. All nodes send 4 messages to node 0 with the order which shown in this figure. If node 0 calls the MPI_Rcv procedure with tag source*j, with source in range {0,..,3} and j in range {0,..,3}, then the matching entry is always at the tail of the Unexpected Queue. In other words, the total number of items traversed is:

$$n + (n - 1) + \dots + 2 + 1 = \frac{(n * (n + 1))}{2}$$

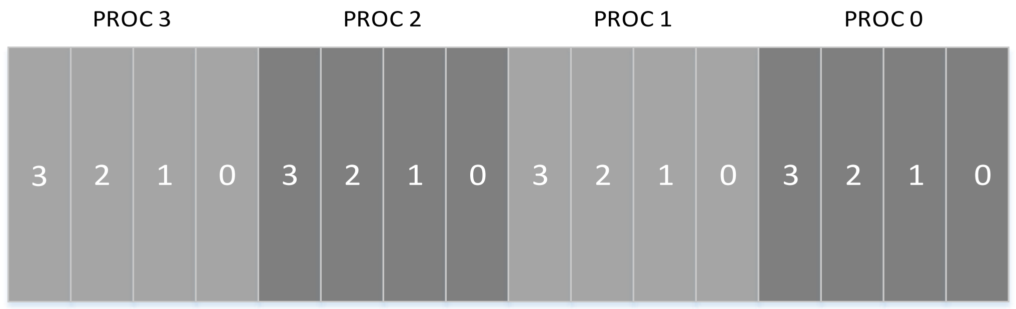


Figure 5.6: Realistic scenario where node 0 has received numerous Unexpected Messages

Figure 5.7 shows our embedded portals software, portals processor in Hardware scheme and a high end host processor with no hashing scheme executing the above realistic scenario. Results show that our Portals Hardware processor is at least one order of magnitude faster than both the high-end processor and the embedded CPU. Initially, high-end CPU processor is faster than embedded Portals Software, but after 16384 messages high end processor suffers from performance degradation due to high number of cache misses. On the other side, for simulated embedded processor we use 1MB cache, while in Portals Hardware we use the same size of cache with 2 ports for read/write for the sake of simulation speed and our aim to get the best possible results.

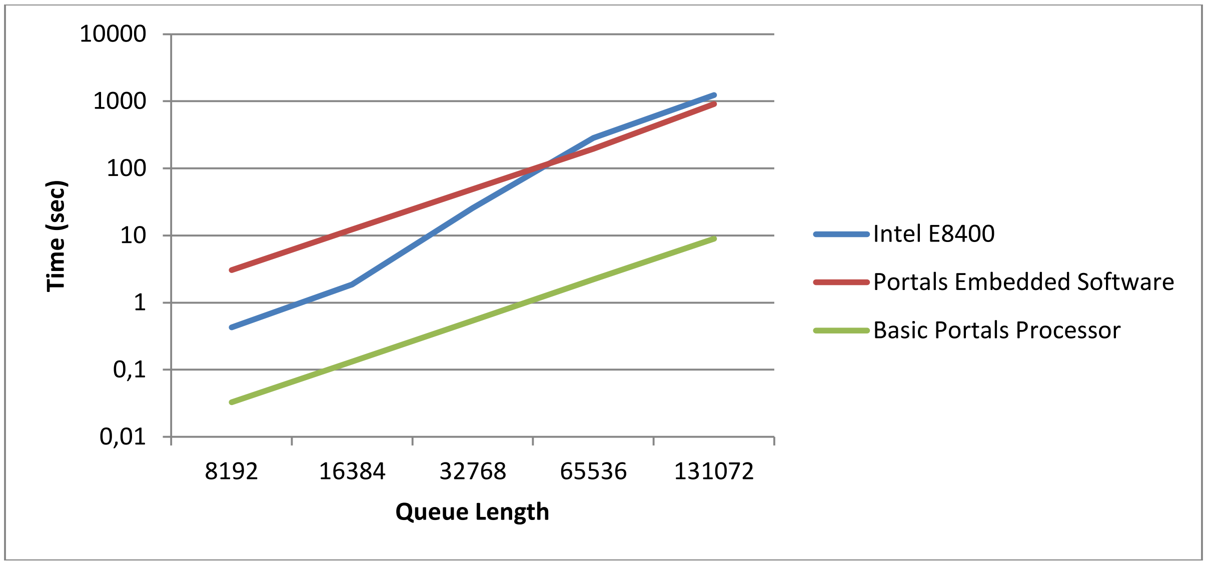


Figure 5.7: Performance of Intel E8400, ARM A9 and Hardware Portals Processor (non-Hashing)

Finally, Figure 5.8 illustrates our Portals Hash-Based Hardware processor is compared with the ARM A9 embedded state of the art CPU executing the above benchmark utilizing the proposed hashing scheme in software. Results demonstrate that Portals Hash-Based Hardware processor is steadily 2 orders of magnitude faster than the embedded CPU.

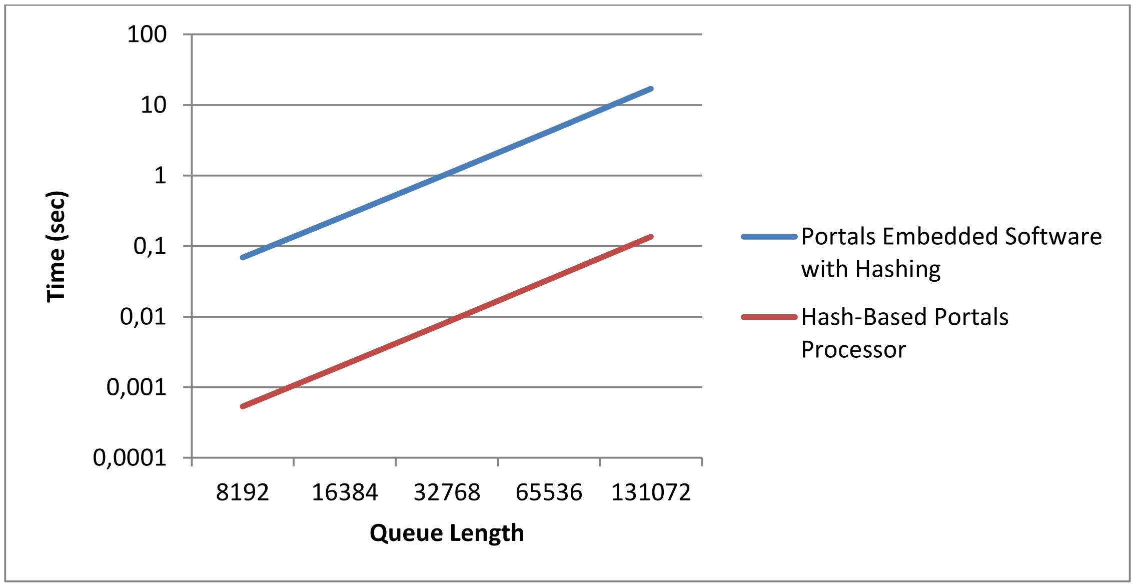


Figure 5.8: Performance of hashed-based ARM A9 and Hardware Portals Processor

5.4 MPI Results

In this section we discuss the performance of our novel approach when executing some of the collective MPI Routines as well as 4 different benchmarks of the NAS benchmark suite. Furthermore, we evaluated the Portals overhead comparing our Portals approach with MPI accelerator in [19]. Finally, we measure the performance of Triggered Rendezvous Protocol.

5.4.1 Results of MPI_Collective Routines

Two of the most commons MPI Collective Routines are MPI_AlltoAll and MPI_AllReduce. These routines were used as a vehicle to examine the performance of our Portals Software implementation running on the embedded processor & Hardware Accelerator. The main reason behind our selection was that these routines use non-blocking Send-Receive commands and there is communication among all nodes.

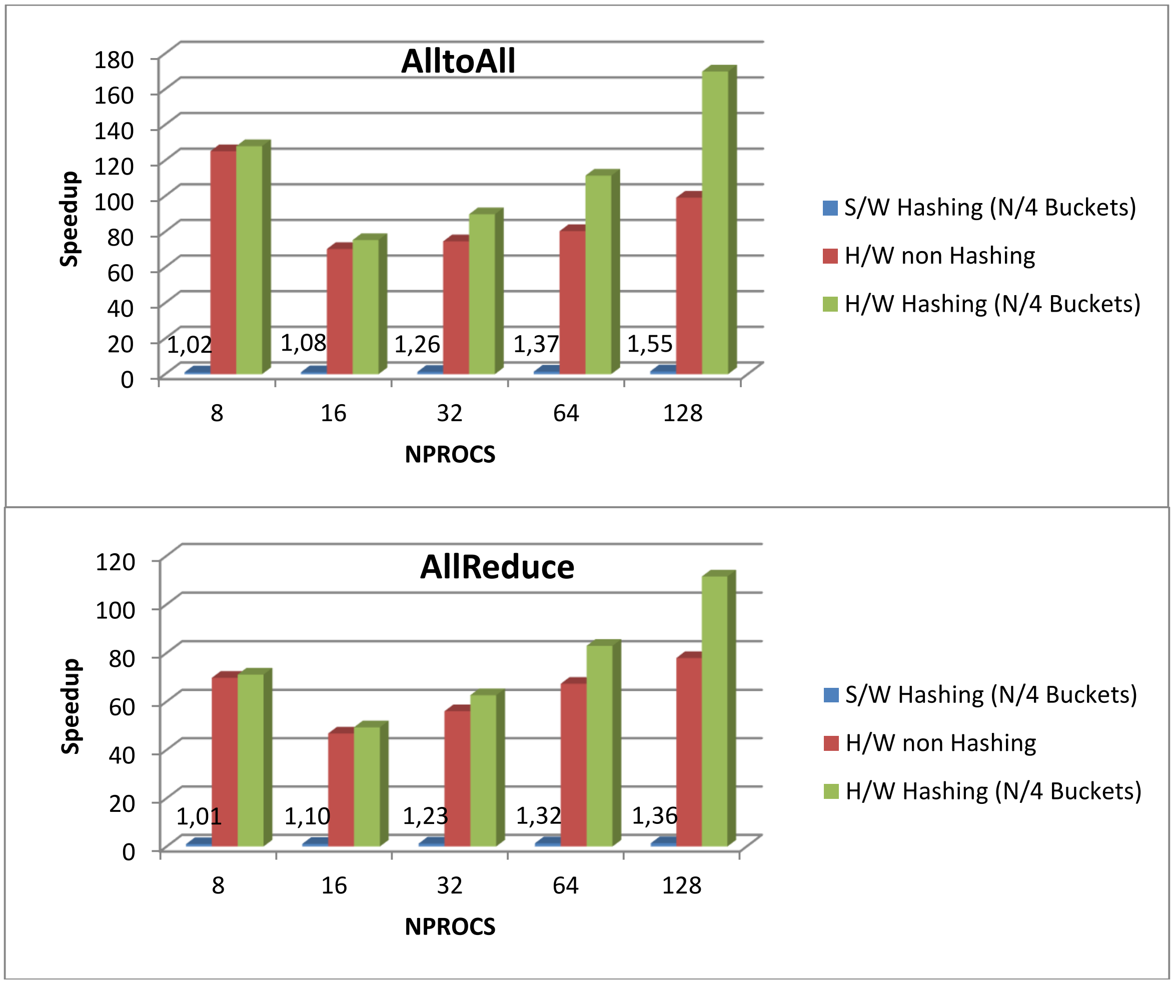


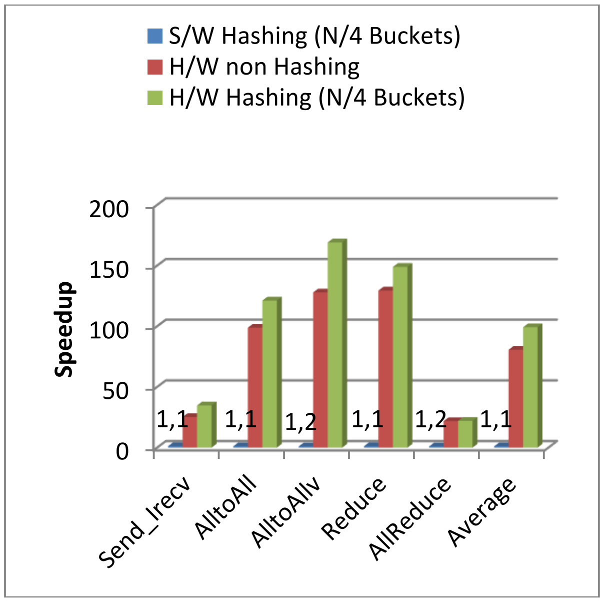
Figure 5.9: Speedup of MPI Collective Routines

Figure 5.9 illustrates the speedup when compared with the ARM A9 embedded CPU, by both the basic hardware device as well as the hash-based module when the above routines are executed on 8,16,32,64 and 128 parallel system with N/4 buckets. S/W hashing scheme has approximately same performance with S/W non-hashing scheme as we measure whole MPI routines (with the idle time which the target waits packets from other nodes (PtlCTWait Routine)) and not only the queue processing speedup as described in [19]. In contrast we achieve significant speedup at H/W Portals Accelerator because we have implemented entirely these routines in Hardware as described in previous sections. Furthermore, H/W

Hashing scheme has significant difference with H/W non-hashing because Hardware ListManager can operate 2 orders of magnitude faster than the S/W ListManager. Finally, we can see that the intercommunication speedup increases with the number of nodes, as a result speedup is expected to be much higher in multi-thousand node systems.

5.4.2 Results of NAS Parallel Benchmarks

We evaluated the performance (in communication routines) and accuracy of our Software and Hardware Platform when executing 4 different benchmarks of NAS Parallel benchmark suite.



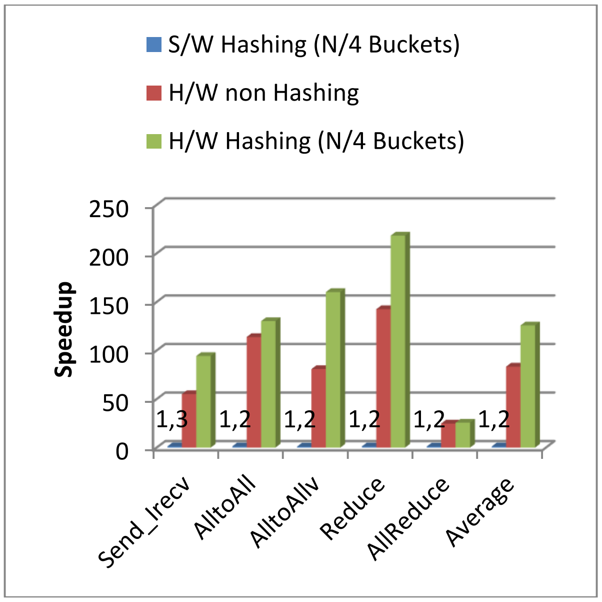
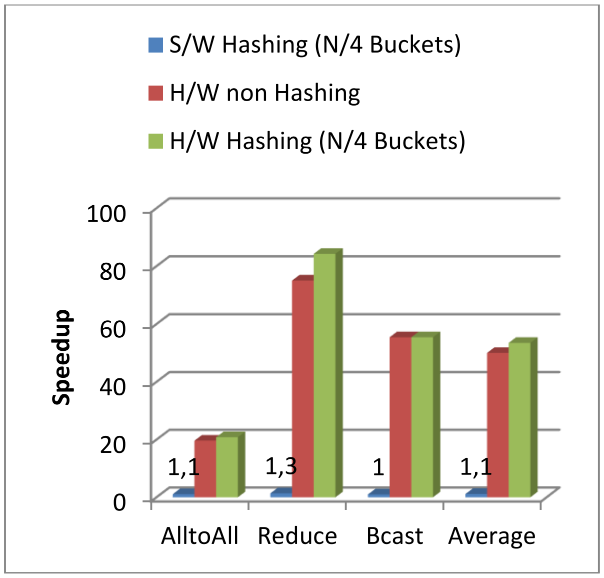


Figure 5.10: Speedup of IS Benchmark (a) 32 nodes (b) 128 nodes



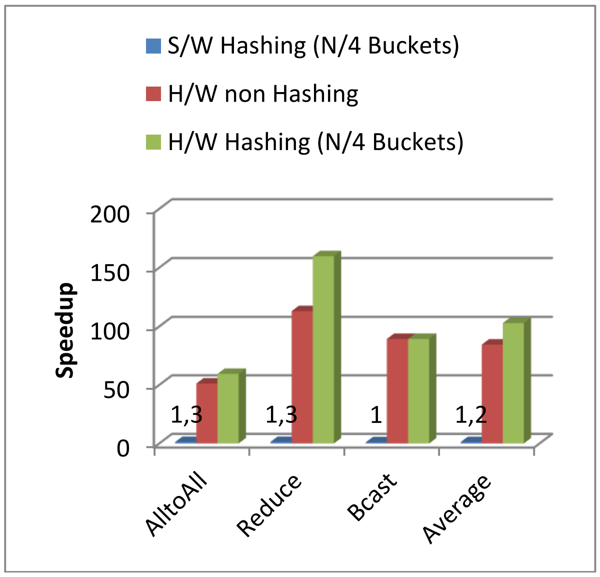


Figure 5.11: Speedup of FT Benchmark (a) 32 nodes (b) 128 nodes

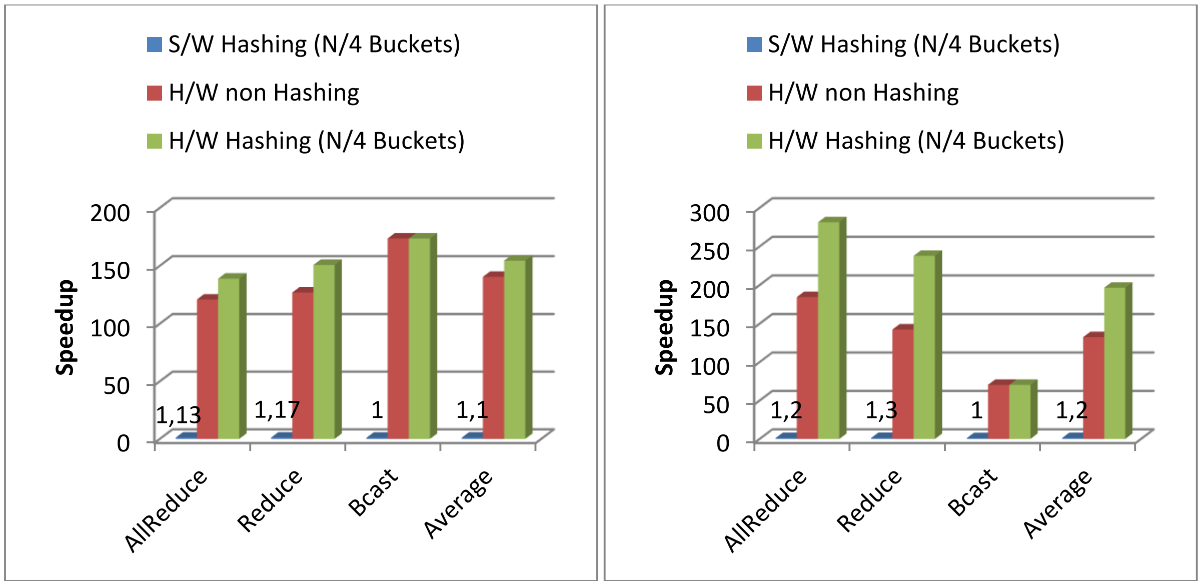


Figure 5.12: Speedup of EP Benchmark (a) 32 nodes (b) 128 nodes

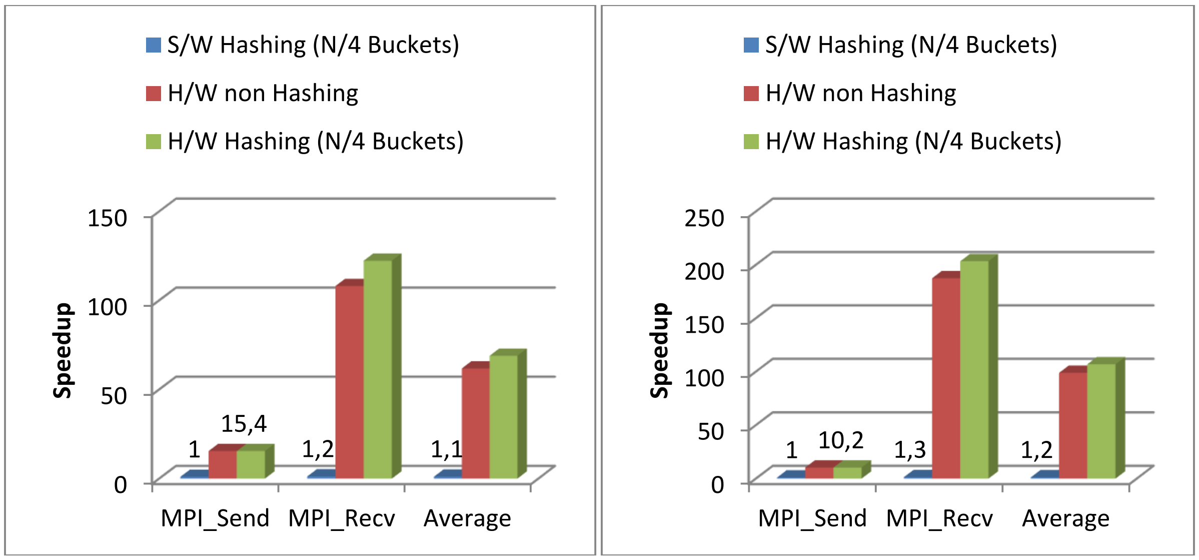


Figure 5.13: Speedup of DT Benchmark (a) 32 nodes (b) 128 nodes

Figures 5.10 - 5.13 demonstrate the speedup triggered by (a) the software-based approach utilizing our hashing function, (b) our Portals hardware accelerator without hashing and (c) our Portals hardware accelerator utilizing our hashing scheme for IS, FT, EP, DT NAS Parallel benchmark within a 32-node platform (left) and 128-node platform (right). Speedup of S/W hash-based approach varies from 10% to over 30%. This speedup is much lower comparing the MPI Accelerator in [19] because we measure whole MPI routines and not only the queue processing speedup. Moreover in Figure 5.13, there isn't any speedup in MPI_Send Routine for the S/W Platform because this Routine simply sends the Data Packet from one node to other without traversing any of the Portals Lists. In contrast, our H/W platform speedup varies from 20x to over 150x for non-hashing and hashing scheme respectively within a 32-node H/W Platform. We can observe that H/W hashing scheme has important speedup comparing non-hashing H/W scheme because all routines' computations have been implemented in H/W. Particularly, all H/W routines computations are translated to specific Hardware modules achieving constant execution time; as a result the performance bottleneck is the List Manager. So, using our hashing scheme is achieved almost constant execution time in collective routines which traverse a significant number of queues, as Reduce, AllReduce, AlltoAll. In contrast in Bcast routine is not achieved speedup with our hashing scheme, because there isn't any queue traversing (one node sends only one message to other nodes). When moving to a larger system, speedup grows as demonstrated in Figures 5.10b - 5.13b; in particular it varies from 20x to over 250x. In multi-thousand node systems this speedup is expected to be much higher [20].

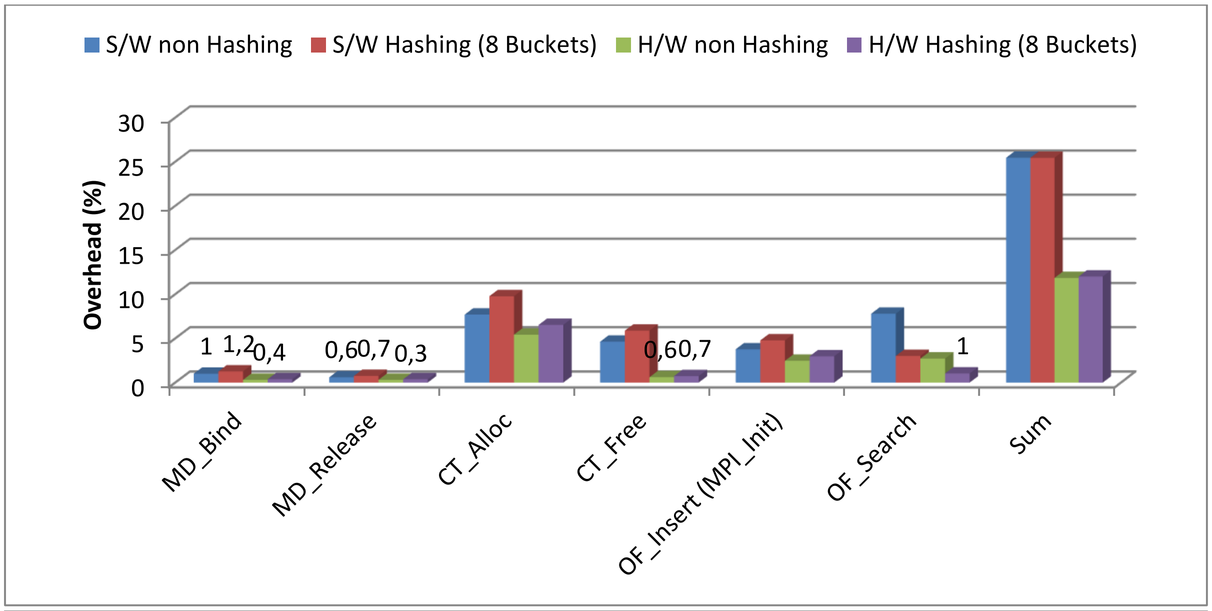
5.4.3 Portals Accelerator versus MPI Accelerator

Portals is a network programming interface which can support both point-to-point interfaces such as MPI as well as the various partitioned global address space models such as PGAS. So, Portals Accelerator implements additional routines comparing with MPI Accelerator as described in [19]. For this reason, we measure the performance overhead of Portals Accelerator with the following procedures[31]:

(i) MD_Bind
(ii) MD_Release
(iii) CT_Alloc
(iv) CT_Free
(v) OF_Insert (MPI_Init)
(vi) OF_Search

Particularly, MD_Bind and MD_Release Portals procedures traverse MD_Table to find the Memory Region in initiator User Space which has been allocated before node begins to send the data packet, while MPI allocates the User Space during MPI_Send routine. Moreover, Portals uses counting events to acknowledge the transaction completion traversing the CT_Table with CT_Alloc and CT_Free procedures, while MPI Accelerator simply waits to receive the Data Packet with MPI_Wait procedure. Finally, Portals accelerator uses Overflow list to save the Payload of Unexpected Messages, while MPI accelerator allocates space for UM payload during MPI Send routine. Figure 5.14 summarizes the overhead (%) per each Portals procedure which differs from MPI Accelerator using MPI AlltoAll routine as benchmark. Simulation time for Bind, Release, Alloc, Free, OF_Insert Routines is the same for both non-hashing and hashing schemes because our hashing scheme isn’t used from these routines.

[31] These routines cause additional overhead comparing with MPI Accelerator.



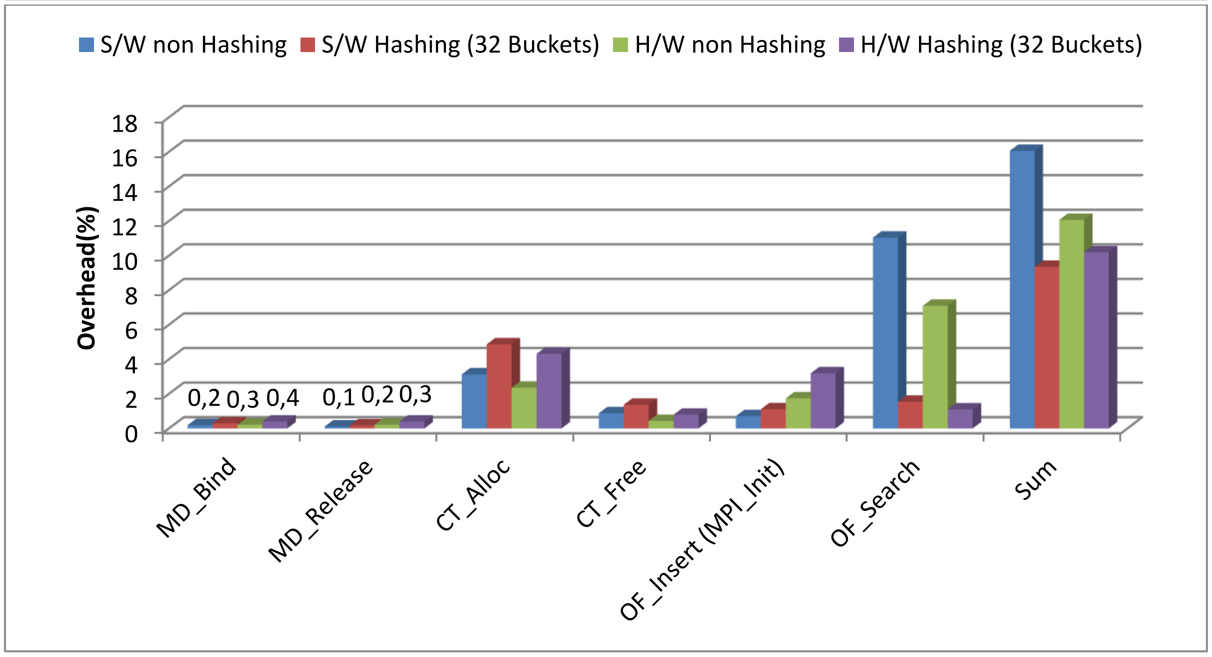


Figure 5.14: Portals-MPI Accelerator Overhead using AlltoAll Routine in (a) 32-node (b) 128-node

In Figure 5.14, overhead in these routines seems to be greater in hashing schemes because total simulation time of AlltoAll Routine in hashing schemes is less than total simulation time in non-hashing schemes. In contrast overhead for OF_Search is much smaller in hashing schemes because it uses our hashing scheme. To summarize, in a 32-node platform Portals Accelerator overhead is approximately 25% and 12% for our S/W and H/W implementation respectively, while in 128-node platform overhead varies from 9% to 15% for S/W implementation and 10% to 12% for H/W implementation respectively.

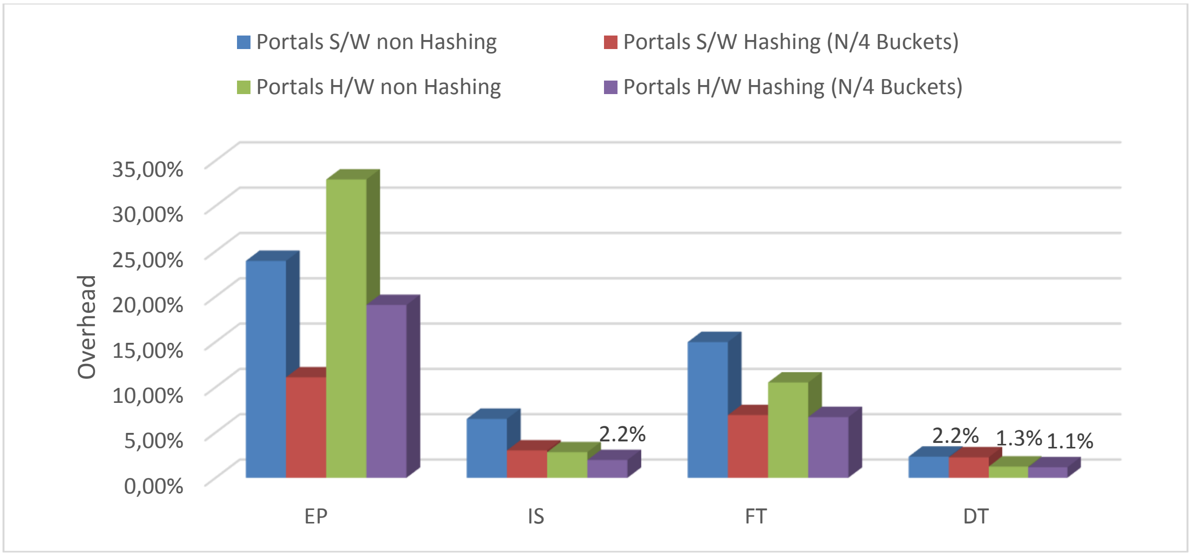


Figure 5.15: Portals-MPI Accelerator Communication Overhead using NAS Benchmarks in 128node

Finally, Figure 5.15 summarizes the communication overhead when executing 4 different benchmarks of NAS benchmark suite. Each application poses different demands with regards to communication of nodes. In EP and FT benchmarks there are more Unexpected Messages during communication process than IS and DT, hence Overflow List must be traversed more frequently in EP and FT than IS and DT, as a result these benchmarks have greater overhead. Finally, DT benchmark has few communication routines and minor Portals overhead.

5.4.4 Rendezvous versus Eager Protocol

In this Chapter we measure the performance of Eager Protocol and Triggered rendezvous Protocol as described in Chapter 4. In Figure 5.16 we use MPI AlltoAll Routine with **8** bytes in each Data Packet (DP). We measure Eager Protocol and two options of Triggered Rendezvous (i) Eager Limit: **0** bytes, and (ii) Eager Limit: **4** bytes. In eager protocol sender simply sends whole Data Packet, while in Triggered Rendezvous with eager limit equal to zero the sender sends only header of DP to receiver, and the receiver issue a get to retrieve the message when the receive is posted. Finally, in Triggered Rendezvous with eager limit equal to 4 bytes, the sender sends header of DP and 4 Bytes to receiver. To summarize, in Eager Protocol there is one transaction (Sender sends whole DP), while in Triggered Rendezvous there are three transactions (i) Sender sends Header and maybe a portion of DP, (ii) Receiver issues a get to retrieve DP, (iii) Sender sends the remainder of DP. In eager Protocol there is a single transaction, but the message might be Unexpected, as a result it is delivered into bounce buffers at any cost[32]. For the same reason the Triggered Rendezvous with eager limit 4 bytes has the worst time, due to the additional Get transaction. The most efficient option is Triggered Rendezvous with eager limit equal to zero, as this protocol is not wasting the bandwidth (if the receiver hasn't space to save the message) and the sender sends Data when

[32] In case of the message is unexpected and it contains payload, both UM & OF Lists are traversed to save the Header and Payload respectively, while in case of the unexpected message not contains payload only UM List is traversed to save the Header of the message.

the receive is posted[33]. In other words, both eager and rendezvous protocols may sends unexpected payloads and as a result these approaches must traverse the OF List to save the unexpected payload, while in rendezvous with zero eager limit only the Unexpected List must be traversed to save the Unexpected Message Header.

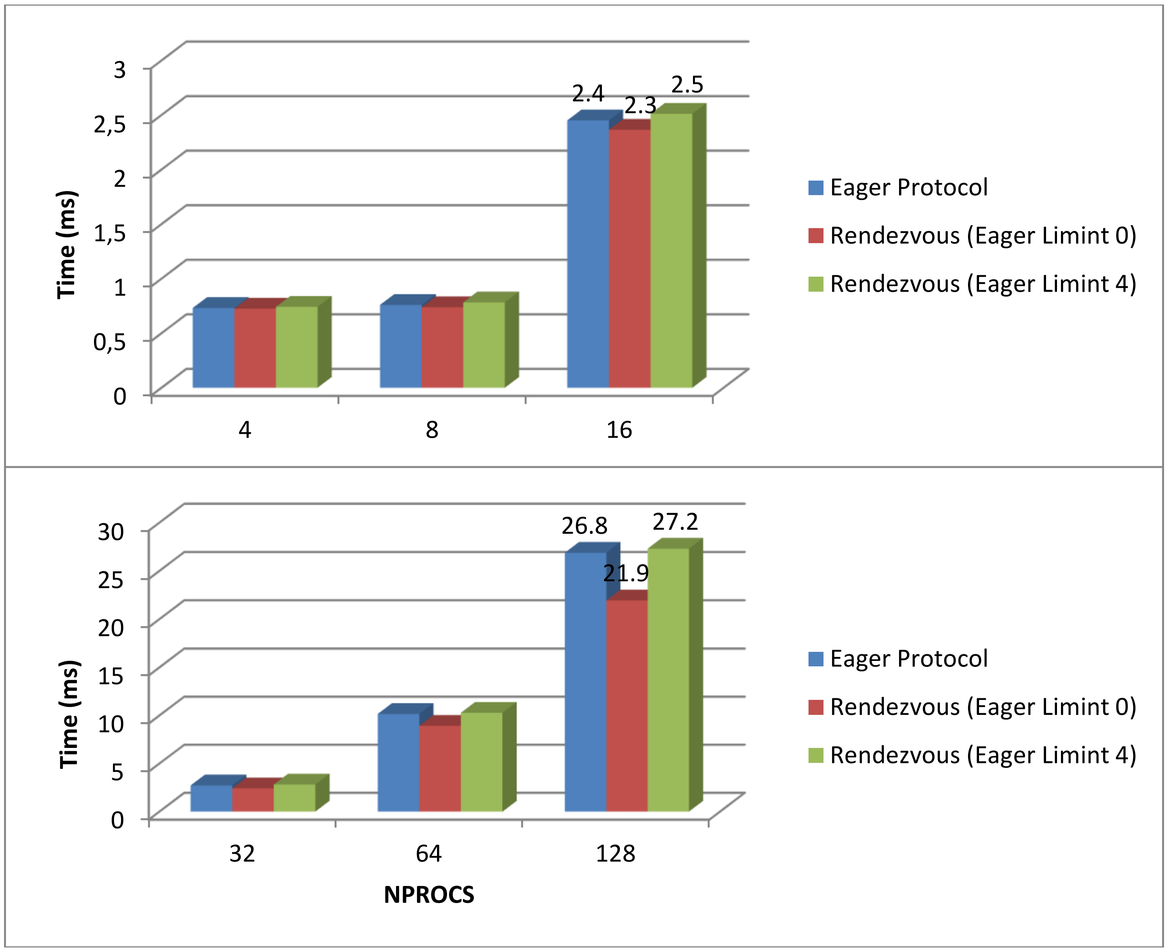


Figure 5.16: Performance of Eager and Rendezvous Protocols

5.5 Global Arrays Results

In this section we discuss the performance and accuracy of our approach when executing some of basic GA Routines as well as one Molecular Dynamic benchmark of Lennard-Jones System. Initially, we implement two benchmarks using GAs Routines to examine the accuracy of our Portals Software & Hardware Accelerator. In the first testbench, addition of two 1-d arrays is implemented[34] as illustrated in Figure 5.17, while figure 5.18 illustrates the speedup of our S/W and H/W Platform in 32-node and 128-node platform.

[33] Triggered Rendezvous with eager limit equal to zero not traverses the OF List in any way, because it not contains Payload with the Header at the initial request.
[34] Each array has NPROCS elements; as a result each thread computes one element.

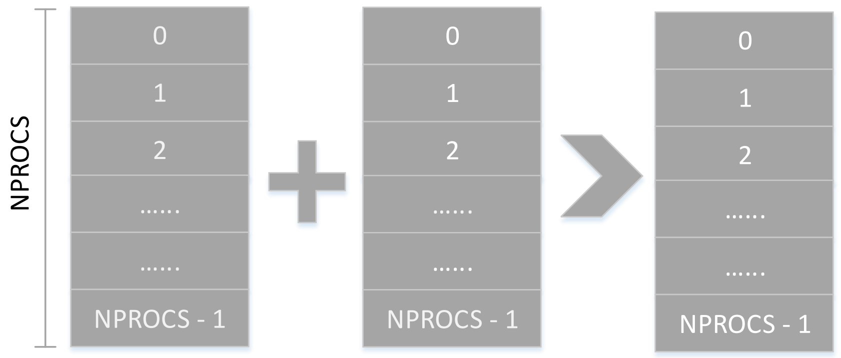


Figure 5.17: Global Array Addition Operation

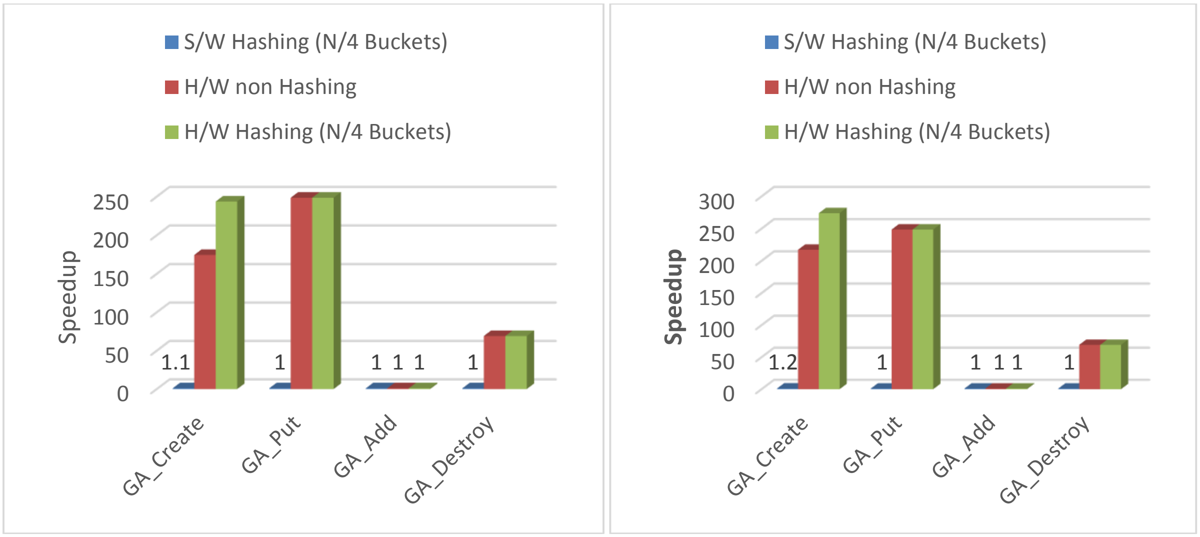


Figure 5.18: Speedup of Addition Benchmark (a) 32 nodes (b) 128 nodes

As described in Chapter 3, in one sided communication we can't use our hashing scheme, because we don't know the source node during GA_Create routine, and as a result GA_Put Routine has the same simulation time in non-hashing and hashing schemes. Furthermore, GA_Add Routine does not need to use any communication routine, as it traverses elements belonging to the same thread. Therefore, the speed depends only to the processor. Unlike GA_Create there is improvement in simulation time using hashing scheme since each node sends information (such as match_bits) to other nodes about the one sided transaction with AlltoAll routine.

In the second benchmark we create and multiply (C=AxB) two 2-d arrays with (NPROCS/4) x (NPROCS/4) elements each one, as shown in Figure 5.19. Each element in C Array needs one row from table A and one column from table B; hence it must issue one Get Operation from A and B arrays. In our benchmark each thread computes $\frac{NPROCS}{16}$[35]elements of table C[36], while each thread needs to get data from (up to $\frac{NPROCS}{4}$) other threads.

[35] With NPROCS>=16.

[36] Tables in Figure 5.18 shows the number of thread which the data belonging.

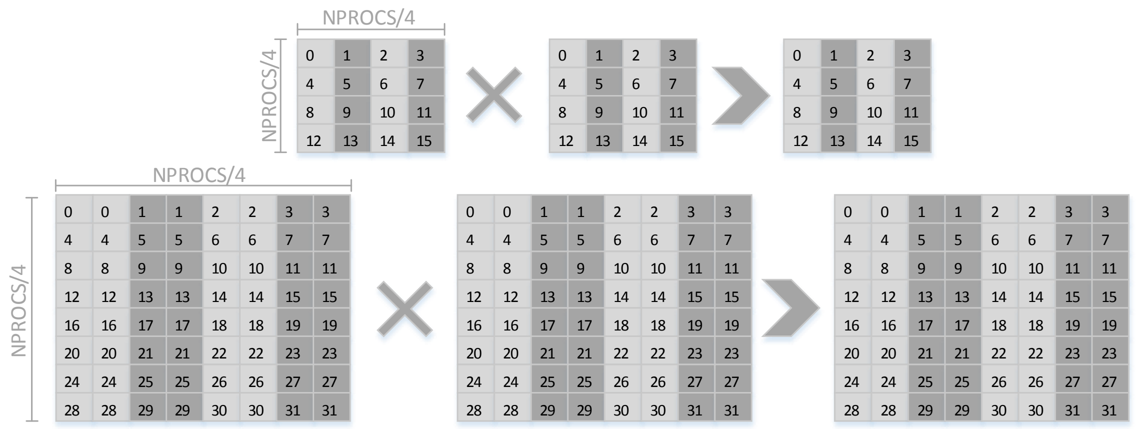


Figure 5.19: Global Array Multiplication Operation (a) 16-node (b) 32-node Platform

So, simulation time quadruples by doubling the nodes as illustrated in Figure 5.20a. Figure 5.20b shows the speedup comparing our S/W and H/W non-hashing scheme[37].

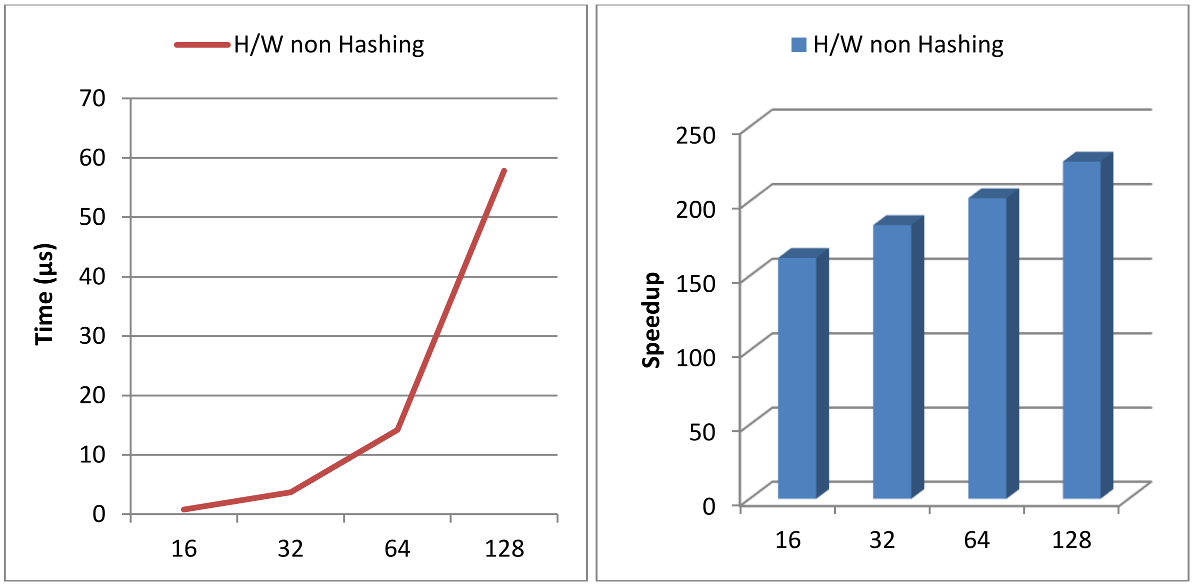


Figure 5.20: (a) Performance (b) Speedup of Global Array Multiplication Operation

Finally, we evaluated the performance and accuracy of our Platform when executing the Molecular Dynamic benchmark of Lennard-Jones System using Global Arrays. Figure 5.21 illustrates the speedup comparing S/W and H/W non-hashing scheme Platforms. Results demonstrate that the speedup of the H/W approach varies from 60x to over 100x, and 100x to over 170x in 32node and 128-node platform respectively.

[37] We don't measure hashing scheme because it has the same simulation time as described in Addition benchmark.

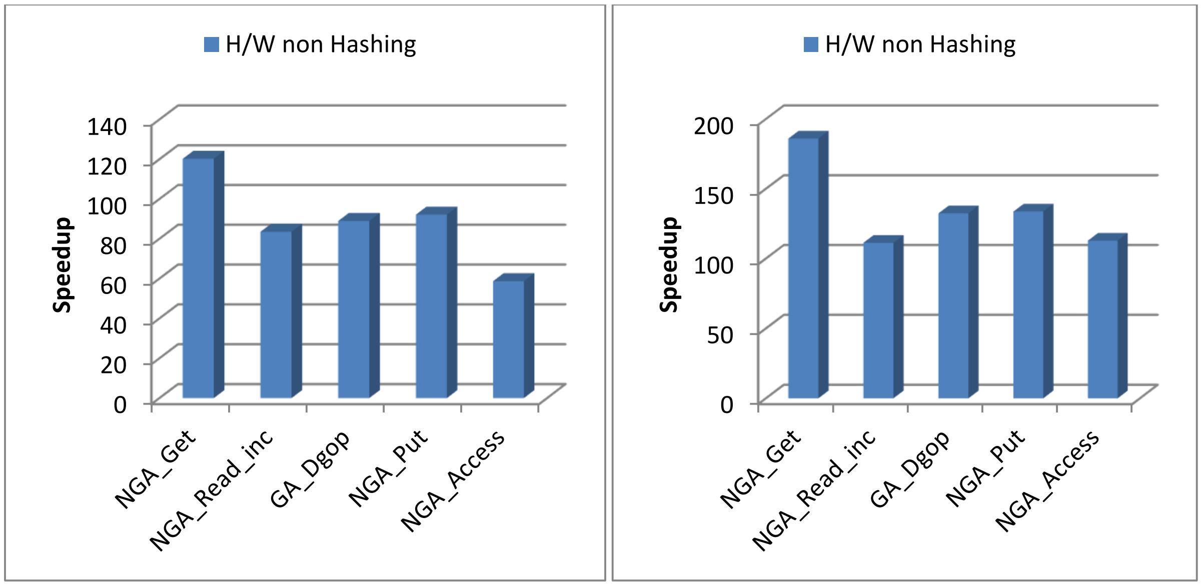


Figure 5.21: Speedup of Molecular Dynamic Benchmark (a) 32 node (b) 128-node Platform

5.6 Area & Power Results

In this chapter we evaluate the Area and Power of our H/W Portals Accelerator and its components (such as List Managers, Memory Allocator etc). The silicon results obtained using Cadence C-to-Silicon high level synthesis tool utilizing 65nm Europractice TSMC standard-cell technology library in Typical Condition (TC).

5.6.1 Area & Power of H/W Accelerator

We evaluate silicon results using Cadence's Incisive Enterprise Simulator (INCISIV) and Encounter RTL Compiler (RC) as described in Figure 5.22[38], while we evaluate SRAM Dynamic Power results using CACTI[39] 6.5. We use MPI AlltoAll Routine as testbench in a 128-node platform for our measurements. Table 5.2 summarizes the Portals Accelerator's SRAMs and shows the Dynamic Power Consumption of each one as computed by the CACTI model. Figure 5.23a illustrates Portals Accelerator Power and SRAMs Dynamic Power using Cadence and CACTI models respectively, while Figure 5.23b illustrates Portals Accelerator Area without SRAMs. Both Power and Area are grown linearly with the frequency, while they are increased only 2,2x and 1,39x respectively tripling the frequency. Based on those figures it is clear that module complexity and power consumption is at least an order of magnitude smaller and lower respectively than that of even a low-power CPU (i.e. ARM Cortex A9 implemented on a 32nm CMOS technology).

[38] TCF is the Cadence standard format to describe switching activity information in a design.
[39] CACTI is an Open Source dynamic power model [20].

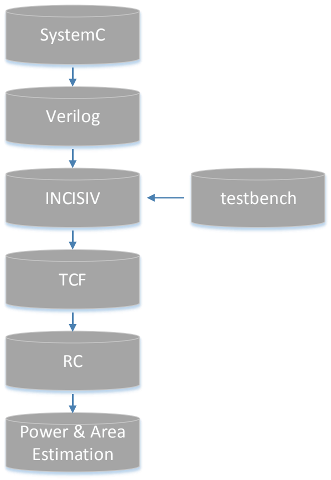


Figure 5.22: Power and Area Estimation flow

SRAM's Description	Capacity	Dynamic Power Consumption
Temporary SRAM for UMs	1M x 8 bits	200 mW/GHz
List's SRAM (PR, UM, OF)	512 x 256 bits	8 mW/GHz
Memory Allocator SRAM	512 x 192 bits	6 mW/GHz
Memory Descriptor Table	256 x 128 bits	2 mW/GHz
Counting Event Table	512 x 128 bits	4 mW/GHz

Table 5.2: Dynamic Power Consumption for SRAMs

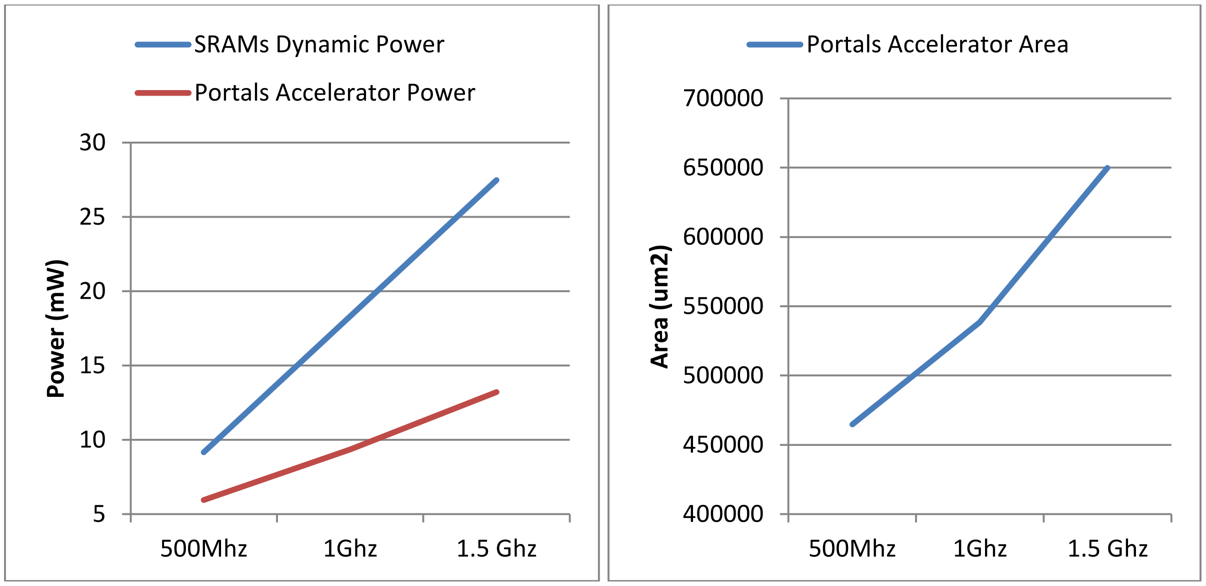


Figure 5.23: (a) Power, (b) Area using AlltoAll benchmark

5.6.2 Results of ListManager Lighter Edition

This section compares the silicon results of List Manager and its Lighter edition. LM Lighter Edition implements one list instead of three lists (PR,UM,OF), as a result the queue processing time is greater than the original List Manager implementation. Figure 5.24 illustrates simulation time with original and lighter ListManager using non-hashing and hashing schemes, while Figure 5.25 compares Portals Accelerator Area and Power using both List Managers.

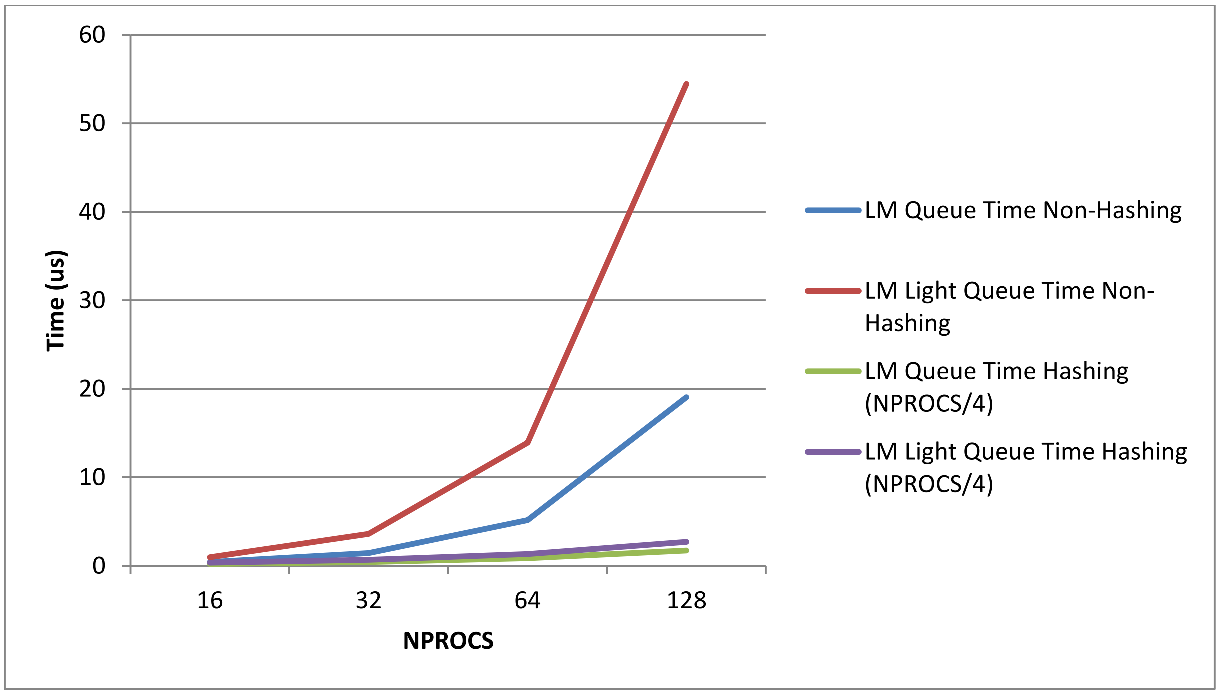
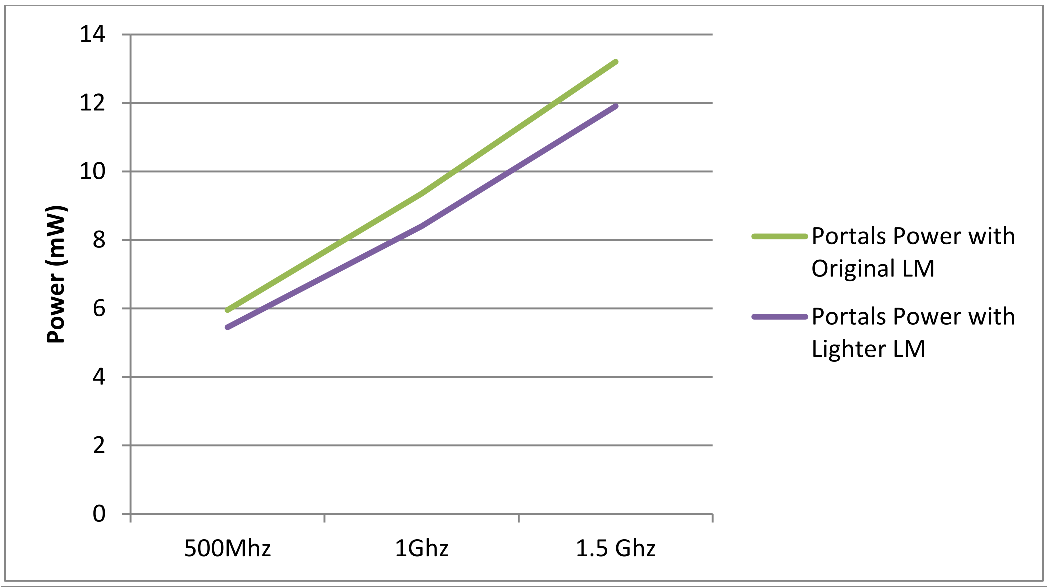


Figure 5.24: Performance of Lighter Edition LM using AlltoAll benchmark

Lighter LM without hashing increases dramatically the queue processing time as the number of processors increases, and as a result it demonstrates suboptimal latency. Portals Accelerator area with Lighter LM has approximately 2,3% less area than Portals with original LM, while power difference increase linearly with the frequency. So, in cases of low power, the Lighter LM with hashing scheme has approximately the same queue processing time, while it can save up to 11% Power and 2,3% Area from Accelerator.



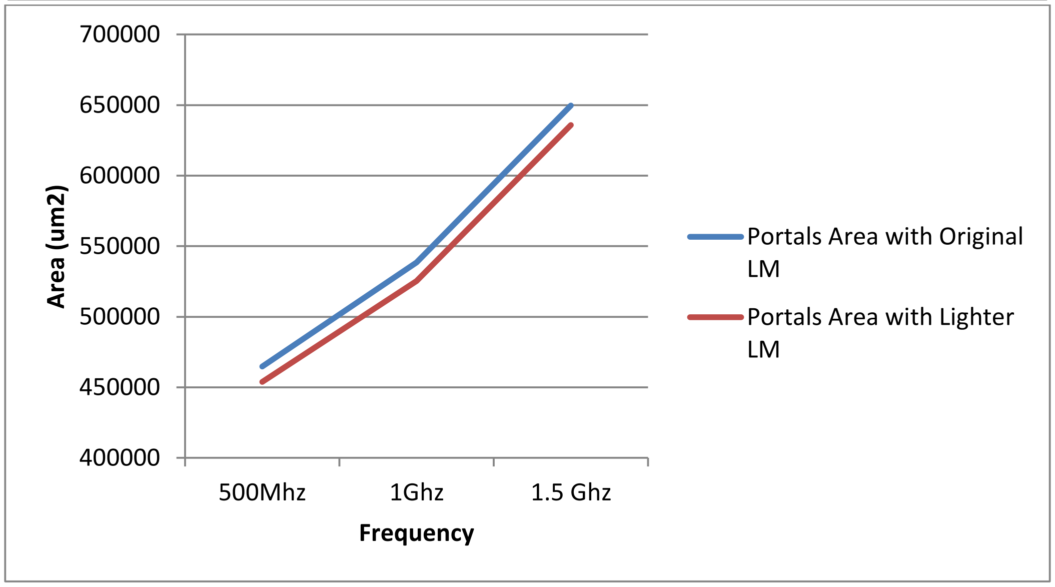


Figure 5.25: (a) Power, (b) Area of Lighter Edition LM using AlltoAll benchmark

6

Conclusions and Future Work

As the number of cores in highly parallel systems increases dramatically, there are a lot of factors which can trigger significant system underutilization. One such underutilization contributor is high intercommunication delay. Although there are several approaches trying to hide delay (such as asynchronous communications primitives), in most of them the processor node should keep track of the status of the various messages sent to and/or received from those millions nodes. This is a time-consuming task, hence offloading it from the main processor, has emerged as an efficient way to reduce intercommunication delay.

This work focuses on minimizing the intercommunication delay of many-core platforms offloading the basic blocks of Portals communication protocol creating the Portals Accelerator. Portals intermediate communication protocol is selected because it provides an interface to support both point-to-point interfaces as well as the various partitioned global address space (PGAS) models. Therefore improving the Portals protocol translates to both point-to-point and PGAS models improvements.

Portals Accelerator in Software is implemented and integrated in an existing multiprocessor framework, so that micro-architectural decisions are based on actual software. The Portals Accelerator is synthesized to actual hardware and thus architectural decisions at the virtual platform level can be rapidly evaluated in terms of area, power and performance. Additionally, a multi-thread environment is implemented which connects each node with its Portals accelerator, and finally, certain tasks of Message Passing Interface (point-to-point) and Global Arrays (PGAS) upper layer Protocols are implemented so as to evaluate the effectiveness and accuracy of our Software and Hardware Accelerator.

Experimental results shows that our Portals Hardware accelerator is from one and up to three orders of magnitude faster than two general-purpose CPUs executing the same tasks, with approximately 15% time overhead comparing with hand-made MPI H/W Accelerator in [19]. Especially, our Hardware Accelerator is up to three order of magnitude faster in processing Portals queues, while it is up to two order of magnitude faster in both MPI and GA Routines with the speedup is grown with the number of nodes in the parallel system. Moreover, our accelerator consumes approximately 100 times less power and it is being implemented at 1/100th of the silicon area of a small embedded CPU. Finally, the remainder Portals operations can be offloaded in the future so as to support more upper layer communication protocols through Portals.

Bibliography

[1] P. M. Kogge et al., “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems,” University of Notre Dame CSE Department Technical Report, TR-2008-13, Tech. Rep., September 28, 2008.

[2] BORKAR, S. AND CHIEN, A. A. 2011. The future of microprocessors. *Commun. ACM 54,* 5, 67–77.

[3] Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. Portals 4 Reference Implementation.

[4]Barrett, B.W.; Brigthwell, R.; Hemmert, K.S.; Pedretti, K.; Wheeler, K.; Underwood, K.D., "Enhanced Support for OpenSHMEM Communication in Portals," *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on* , vol., no., pp.61,69, 24-26 Aug. 2011

[5] http://en.wikipedia.org/wiki/Partitioned_global_address_space

[6] Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. Portals 4.0 Specification. April 2008

[7] Brian W. Barrett, Ron Brightwell, K. Scott Hemmert, Kyle B. Wheeler, Keith D. Underwood. Using Triggered Operations to Offload Rendezvous Messages. EuroMPI 2011, pages 120-129.

[8] https://code.google.com/p/portals4/downloads/list

[9] http://www.ovpworld.org/

[10] Manolis Katevenis, Link and Memory Architectures and Technologies. Lecture in CS-534 – Univ. of Crete and FORTH, Greece page 10

[11] KNUTH, D. E. 1998. The art of computer programming, volume 1 (3rd ed.): Fundamental Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Jeffrey Daily, Abhinav Vishnu, Bruce Palmer, Hubertus van Dam. PGAS Models Using an MPI Runtime: Design Alternatives and Performance Evaluation. SCHEDULE: NOV 16-22, 2013.

[13] Cadence C-to-Silicon Compiler High-Level Synthesis datasheet

[14] K. Georgopoulos, Ioannis Papaefstathiou. A Concise Review of HLS Tools and Compilers. July 2014

[15] Cadence Cadence C-to-Silicon Compiler User Guide. Product Version 11.10 s200, August 2011

[16] http://www.eejournal.com/archives/articles/20131031-cadence/

[17] D. Bailey, E. Barszcz, J. Barton, D. Browning. THE NAS PARALLEL BENCHMARKS. RNR Technical Report, March 1994.

[18] E. R. Hernández. Molecular Dynamics: from basic techniques to applications. Institut de Ciència de Materials de Barcelona

[19] Pavlos Mattheakis, Ioannis Papaefstathiou. Significantly Reducing MPI Intercommunication Latency and Power Overhead in Both Embedded and HPC Systems. ACM Transactions on Architecture and Code Optimization (TACO), January 2013

[20] BRIGHTWELL, R. AND UNDERWOOD, K. D. 2004. An Analysis of NIC Resource Usage for Offloading MPI. Parallel and Distributed Processing Symposium, International 9.

[20] http://www.hpl.hp.com/research/cacti/