Accelerating Intercommunication in Highly Parallel Systems

NIKOLAOS TAMPOURATZIS and PAVLOS M. MATTHEAKIS,

Technical University of Crete, Greece

IOANNIS PAPAEFSTATHIOU, Synelxis Solutions Ltd

Every HPC system consists of numerous processing nodes interconnect using a number of different inter-process communication protocols such as Messaging Passing Interface (MPI) and Global Arrays (GA). Traditionally, research has focused on optimizing these protocols and identifying the most suitable ones for each system and/or application. Recently, there has been a proposal to unify the primitive operations of the different inter-processor communication protocols through the Portals library. Portals offer a set of low-level communication routines which can be composed in order to implement the functionality of different intercommunication protocols. However, Portals modularity comes at a performance cost, since it adds one more layer in the actual protocol implementation. This work aims at closing the performance gap between a generic and reusable intercommunication layer, such as Portals, and the several monolithic and highly optimized intercommunication protocols. This is achieved through the development of a novel hardware offload engine efficiently implementing the basic Portals' modules. Our innovative system is up to two orders of magnitude faster than the conventional software implementation of Portals' while the speedup achieved over the conventional monolithic software implementations of MPI and GAs is more than an order of magnitude. The power consumption of our hardware system is less than 1/100th of what a low-power CPU consumes when executing the Portal's software while its silicon cost is less than 1/10th of that of a very simple RISC CPU. Moreover, our design process is also innovative since we have first modeled the hardware within an untimed virtual prototype which allowed for rapid design space exploration; then we applied a novel methodology to transform the untimed description into an efficient timed hardware description, which was then transformed into a hardware netlist through a High-Level Synthesis (HLS) tool.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures; C.3 [Special-Purpose and Application-Based Systems]: Real-Time and Embedded Systems

General Terms: Architecture, Design, Performance

Additional Key Words and Phrases: Portals, MPI, GA, HPC, intercommunication, embedded systems

ACM Reference Format:

Nikolaos Tampouratzis, Pavlos M. Mattheakis, and Ioannis Papaefstathiou. 2016. Accelerating intercommunication in highly parallel systems. *ACM Trans. Archit. Code Optim.* 13, 4, Article 40 (December 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/3005717>

1. INTRODUCTION

Achieving an exascale level of performance requires several fundamental changes in hardware and software that will affect all areas of high-performance computing [Bergman et al. 2008]. In that respect, systems with numerous CPUs, along with

Authors' addresses: N. Tampouratzis and P. M. Mattheakis, Telecommunication Systems Institute, Technical University of Crete, Kounoupidiana, GR73100, Chania, Greece; emails: ntampouratzis@isc.tuc.gr, pmat@csd.uoc.gr; I. Papaefstathiou, Synelxis Solutions Ltd., 10 Farmakidou St., Chalkida, GR34100, Greece; email: ygp@synelxis.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/12-ART40 \$15.00

DOI: <http://dx.doi.org/10.1145/3005717>

several customized accelerators, have been recently proposed as, probably, the only viable solution offering high performance at a low-energy budget [Borkar and Chien 2011].

Unfortunately, in highly parallel systems with a very large number of cores, there are several factors that reduce the system's utilization significantly. One of the main ones is that during inter-communication, cores may remain idle for synchronization purposes. Increasing the number of cores in a system also increases the time spent in communication, as each core has less work to do and more cores with which to communicate. Hence, as the number of cores increases, a crucial challenge is to reduce the overhead of the inter-processor communication.

During the last decades, several research groups have proposed certain communication layers such as UCX [UCX 2016], OFI [Choi et al. 2016], and Portals [Barrett et al. 2014] in order to homogenize the intercommunication mechanisms in parallel systems. UCX is based on a recent collaboration between industry and academia in order to create an open-source production-grade communication framework for data-centric and high-performance applications, while OFI is a user-level API for network programming, providing a standardized interface for higher-level clients such as MPI and PGAS [PGAS 2013]. Moreover, Portals is an intermediate communication layer that allows scalable, high-performance intercommunication between nodes in a parallel computing system.

In our work, we focus on Portals since it is currently the most widely used such protocol. Specifically, Portals are based on the concept of elementary building blocks that can be combined to support a wide variety of higher-level point-to-point and distributed shared memory (such the partitioned global address space (PGAS)) protocols. In addition, several inter-communication protocols that can be implemented on top of Portals (e.g., MPI) have been accelerated through co-processors [Brightwell et al. 2005], optimized embedded software on NIC's processors [Brightwell and Underwood 2004] or even with dedicated hardware [Mattheakis and Papaefstathiou 2013]. Regarding the acceleration of the Portals layer itself, there are two approaches that have been proposed so far: the first offloads most of the Portals processing to the PowerPC of a Network Interface Card (NIC) [Brightwell et al. 2006], while in the second a highly specialized multicore Application Specific Instruction set Processor (ASIP) was developed that supports an instruction set that was designed specifically for the Portals scheme [Derradji et al. 2015]. In this article, we present the very first tailor-made SoC-oriented hardware system that accelerates significantly the execution of the Portals layer.

Designing a specialized accelerator of a complex software, such as Portals, is not a trivial task, as both hardware and low-level software have to be developed; while the software development 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 utilized so as to offer to the software developers a virtual model of the hardware platform at the very early stages of the product development. Virtual models usually support numerous popular CPUs (such as MIPS, ARM, PowerPC) along with widely used peripheral models, such as USB, DMA controllers, and the like; those processors together with their peripherals can be co-simulated with the under-development software at a very early stage.

The contribution of the proposed work can be summarized in the following points.

- The first known novel SoC acceleration core for the Portals library (called PAC) is analytically demonstrated implementing the Portals API, which is specifically designed as hardware-friendly intercommunication layer;
- An innovative design path to actual hardware implementation is presented starting from virtual prototypes and ending at silicon;

Fig. 1. (a) Blocking receive; (b) Non-blocking receive commands.

- Directions are provided for fine tuning higher level hardware descriptions leading to efficient implementations;
- Several widely used benchmarks are ported to this new system and evaluated across different metrics.

The rest of the article is organized as follows. Section 2 reviews Portals. Section 3 presents the architecture of our system as it is modeled within a state-of-the-art virtual platform. Section 4 presents our novel approach for rapidly transforming the architectural description into a timed hardware description, and for verifying the latter; our approach takes advantage of the recently introduced High-Level Synthesis (HLS) Tools. Then, Section 5 presents real-world performance, power and silicon area results, based on several benchmarks. Finally, Section 6 concludes this work.

2. BACKGROUND AND PORTALS API

For nearly 20 years, the Portals intermediate layer was being developed by Sandia National Laboratories and the University of New Mexico. It enables scalable, high-performance network communication between nodes in a parallel system. This section initially introduces the concepts of one-sided, two-sided, blocking, and non-blocking communication operations in Portals. Then, it presents an overview of Portals basic data operations as well as the operation of the Portals' Lists; and finally the completion event mechanism, the memory descriptors, and the rendezvous protocols are analyzed.

2.1. One-Sided, Two-Sided, Blocking and Non-Blocking Communication

In two-sided communication, both the sender and the receiver require an implicit synchronization scheme within which the messages are matched based on message identifiers. 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. Portals provides both two-sided and one-sided data movement operations so as to allow for the implementation of certain protocols that require both of them such MPI and PGAS.

Communication typically consumes a significant part of a parallel application's execution time. During communication, the actual computation is blocked, unless an overlapping of these two is supported; this overlapping is described in Figure 1 in which processor 1 (P1) sends a message to processor 2 (P2). In Figure 1(a), if the message has not yet arrived when P2 executes a blocking receive command, then P2 is blocked until the actual message arrives. On the other hand, in the non-blocking scenario shown in Figure 1(b), P2 resumes computation after executing a non-blocking receive command.

In order to implement the above communication protocols, portals use mainly two basic data movements operations: *PtlPut* and *PtlGet*. In *PtlPut*, the sender sends one

(or more) data packet(s) to the receiver, while in *PtlGet* the node (which requires data from some other node) sends its header (*PtlGet*) to some other node and then it waits for the corresponding response.

Every data movement operation involves two processes (i.e., nodes); in Portals' terms, these are the *initiator* (Sender) and the *target* (Receiver). The initiator initiates the data movement operation, while the target responds to the operation by accepting the data of a put operation, or replying with the data in a get operation.

Another important Portals routine is the *PtlMEAppend*¹ which is used by the target node when it receives (in *PtlPut*) or sends (in *PtlGet*) message payload in two-sided communication.

2.2. Portals Lists

Portals uses three message lists at the target node: Priority (PR), Unexpected (UM), and Overflow (OF). The PR list stores the header of the expected messages, the UM list stores the headers of the unexpected messages, while the OF list stores the data of the small unexpected messages.

Both the UM and the OF lists grow significantly in programs that use eager *Put* commands and in which the sender assumes that the receiver has enough space to buffer the headers and payloads, while the size of the PR list depends on the number of receive commands. At message arrival, the PR list is first processed and, if no matching entry is found, then the OF list is processed so as to find some available space for the message payload. If there is available space, a message payload is delivered into the OF list and its header is placed into the UM list. On the other hand, when a new entry is appended to the PR list, the UM list is first searched for a match. If a match is found, the header is removed from the UM list, and the application is notified that there is a match.

The size of the three message lists severely affects the performance of Portals, as applications may traverse a significant number of entries searching for a certain message. Hence, accelerating list processing will result in a significant increase in the performance of Portals. In order to look deeper into the list handling in Portals, Figure 2(a) describes in detail the *PtlPut* operation executed by P2 during a non-blocking² receive such as the one in Figure 1(b). In the abstract view (Figure 1), it was assumed that a non-blocking receive call returns as soon as the receive command is posted. Looking at the scenario in more detail, a number of operations should be performed in order to execute a complete receive command. When the target wants to receive one message from another node, it calls the *PtlMEAppend* routine to append a receive request. During this process, it must first search the UM list in case this message has already been received. Accordingly, when the message arrives in P2, the computation phase 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. If there are no matching packets in the PR list, it traverses the OF list to find sufficient space to save the unexpected message. *PtlCTWait* and *PtlEQWait* are the Portals' blocking commands which wait for the message arrival; the *PtlCTWait* command is used in this work due to its lightweight characteristics, which allows for faster hardware implementation. In Figure 2(a) UM, PR, and OF should all be searched and the *PtlCTWait* command should be executed 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 and the communication stages as illustrated in Figure 2(b).

¹*PtlMEAppend* is not officially part of the Portals 4.0.2 specification.

²All data movements in Portals are non-blocking (such as those in *PtlPut*, *PtlMEAppend*) except the ones initiated *PtlCTWait*.

Fig. 2. (a) CPU non-blocking receive; (b) Offload non-blocking receive schemes.

2.3. 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 identification of 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).

2.4. Portals Memory Descriptors

Memory descriptors (MDs) are initiator side resources that are used to encapsulate the association of a network interface (NI) with a memory region. They provide an interface to register memory spaces and to carry that information across multiple operations (an MD is persistent until released). The *PtlMDBind* command creates a memory descriptor and *PtlMDRelease* unlinks and releases the resources associated with a memory descriptor.

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 certain scenarios [Barrett et al. 2011a]. The typical approach is to use the Portals eager protocol for short messages and switch to a rendezvous protocol for long messages. This subsection presents the implementation of both eager and rendezvous protocols in Portals.

The Portals eager protocol sends whole messages including Header and Payload eagerly, as illustrated in Figures 3(a)–3(b). If a message is expected,³ it is delivered directly into the user space of the process and an ack is (optionally) generated to notify the sender that the message was successfully delivered (Figure 3(a)). On the other hand, if the message is unexpected, there are two different cases. In the first scenario

³When *PtlMEAppend* has been issued by the target node before the message arrives, the message is expected; else the message is unexpected.

Fig. 3. Communication pattern for Portals eager protocol and rendezvous protocols.

(Figure 3(b)), the receiver discards the payload of the message while it keeps the Header in order to issue a get request to retrieve the payload when a *PtlMEAppend* command is issued. In the second scenario, the target node has sufficient bounce buffer to save the payload so the initiator should not retransmit the payload when needed (i.e., a get request is not issued from target node).

In the case of the Rendezvous protocol, the initiator only sends a piece of the message (the size is determined from *eager_limit* variable) containing sufficient information in the header for the *PtlMEAppend* command to issue a get operation to retrieve the message (Figures 3(c)–(d)). 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 (OF list). If the total size of the message is greater than the *eager_limit*, the target issues a *PtlGet* command to retrieve the remainder of the payload.

3. SYSTEM ARCHITECTURE

This section presents the architecture of our novel Portals acceleration system (PAC). This architecture has been developed and evaluated on top of the Open Virtual Platform (OVP) [OVP 2014] environment. OVP aims at reducing the product development time, especially for MPSoC platforms, by giving the software developers a virtual model of the hardware platform at the very early stages of the development phase. OVP supports several popular CPUs (such as MIPS, ARM, PowerPC) along with widely used peripheral models, such as USB, DMA controllers etc. In this work, we selected the two most widely utilized CPUs, namely a high-performance Intel one and a

Fig. 4. Application and system engineers view of the platform modeling and simulation flow.

low-power ARM CPU, and we compare their performance and power consumption when executing the portals code, with those achieved by PAC.

OVP is supported by an instruction-accurate and fast simulator, OVPsim. OVPsim provides the infrastructure for simulating platforms with one or more processors containing shared memory and busses in arbitrary topologies connected to different peripheral modules. The performance of OVPsim depends on several factors (such as the processor type, the complexity of the application, and the like), but typically it can handle hundreds of millions of simulated instructions per second.

Figure 4 illustrates the flow for obtaining a simulation executable that encapsulates not only the parallel application but also a high-level model of the complete hardware platform. The parallel application is compiled and loaded in the memories of the CPUs of the platform. The platform and the custom hardware are described using the OVP Innovative CPU Manager (ICM)⁴ and Behavioral Hardware Modeling (BHM)⁵ APIs, respectively [Claesen et al. 2015]. The complete system can be simulated by the OVPsim exercising actual software on under development hardware platforms.

3.1. High-Level Architecture

This section demonstrates our high-level architecture while it also lists our approach for modeling this architecture within OVP. Initially, we selected the Open Risc 1000 (OR1K) processor as the processing core of each node in our parallel system. It should be noted that during system's simulation, each of the OR1K-based nodes is simulated in the host machine at predefined time periods. In more detail, the simulator calculates the number of instructions that should be executed by a processor in a time slice, and then it simulates those instructions. At the end of the time slice, the simulation of this particular processing node is suspended and the next node is simulated for another time slice. This is a pseudo-parallel approach that emulates the concurrent behavior of an actual multi-processor platform.

An overview of our high-level architecture is presented in Figure 5, which also highlights how this architecture is modeled on top of OVP. It should be stressed that in

⁴OVP ICM is an application used to instantiate, interconnect and simulate complex multiprocessor platforms composed of processors, memories and busses.

⁵BHM functions permit the definition behavioral models of hardware and software components that act as peripherals to the processors.

Fig. 5. OVP platform with our Portals accelerator.

OVP any intercommunication between the nodes is modeled through memory reads and writes. Our reference platform consists of N processing nodes and each node is comprised of one OR1K processor, a local memory, and one PAC model connected with a bus. In turn, our PAC module contains a dynamic memory allocator for the unexpected Portals messages, a list manager for the processing of the different lists and an accelerator buffer. Furthermore, Figure 5 shows the space reserved for the modeling within OVP of the memory-mapped intercommunication. The memory space that corresponds to PAC is common for all processors as one global address space is used for *PtlPut* and two global address spaces for the *PtlGet* transaction. In a *PtlPut* transaction, the initiator node writes the message to the appropriate global space in ‘*PUT GLOBAL ADDRESS_SPACE*’, while the target node can read the message from the same space. In a *PtlGet* transaction, when the initiator node writes the header of the message request in the ‘*GET HEADER GLOBAL ADDRESS SPACE*’, the target reads the header from this same space. Later on, the target, in turn, responds to an initiator request with the message Payload which is placed in the ‘*GET PAYLOAD GLOBAL ADDRESS SPACE*’.

In other words, there is a unique global address scheme which associates a certain memory space with the PAC's buffer of a specific node. We use three distinct global address spaces and we cannot share the two spaces corresponding to the message payloads (i.e., `'PUT_GLOBAL_ADDRESS_SPACE'` and `'GET_PAYLOAD_GLOBAL_ADDRESS_SPACE'`) because one node may issue *PtlPut* and *PtlGet* operations at the same time to some other node (i.e., in Rendezvous Protocol).

3.2. PAC's Micro-Architecture

The framework presented in Section 3.1 allows the modeling of our under-development hardware within a complete parallel system. For the description of the hardware, we

Fig. 6. Portals Accelerator Processor (PAC).

used the untimed functional abstraction level as described in Cai and Gajski [2003] in which the low-level timing details are hidden so as to accelerate the simulation execution time.

The basic components of our novel PAC system are shown in Figure 6. The functionality of each component is analyzed in the following subsections.

3.2.1. Master/Slave Ports. PAC is connected to the virtual bus through master and slave ports, depending on whether it initiates or responds to bus transactions. In order to connect PAC to OVP's virtual bus, we utilized both the Innovative CPU Manager (ICM) API for the Global Address spaces and the Peripheral Programming Model (PPM)⁶ API for the open, close, read and write operations of the Master and Slave Ports [Claesen et al. 2015]. Specifically, three master ports are handling the read and write of the Portals messages to the other nodes, while the operations of each accelerator are triggered by three distinct OVP events (i.e., one for each incoming message type) using three slave ports.

3.2.2. Message Buffers. Three Message Buffers are placed at three corresponding slave ports triggering the accelerator when three different types of incoming Portals messages arrive. These buffers store the requests from the processor, which are effectively, simple memory writes to appropriate addresses. In this way, the overhead of the host-processor is minimal, since it can return to its normal operations just after it writes the Portals command to the message buffer. The master ports, on the other hand, do not need buffers as the headers and the payloads of the Portals messages are stored in the corresponding global memory spaces. In our case, we use a Message Buffer of size $NPROCS * MAX_MSG_SIZE$ ⁷ bytes; when the message buffer gets full an interrupt is raised to the processor.

3.2.3. Portals Message Processor. The Portals Message Processor (PMP) orchestrates the data flow through all PAC's components based on the control flow imposed by each Portals command. Initially, when a request message is placed at the message buffer, PMP decodes the '*command type*' and identifies the message (Header and possibly Payload) position in the Global Address Spaces. Hence, besides the message buffers, PMP communicates with the master ports that have access to the Global Address Spaces as well as with the Accelerator's buffers. Moreover, PMP issues certain list

⁶OVP PPM is an application modeling the interfaces and the connections (bus ports, and the like) to the simulated platform.

⁷ MAX_MSG_SIZE is fully configurable at design time.

Fig. 7. Portals list manager.

commands to the list manager based on the message status while it requests memory space from the dynamic memory allocator in order to save the unexpected messages. Each operation (Portals command) is implemented in a different PAC subsystem and as a result different Portals operations can be executed concurrently.

3.2.4. List Manager. The list manager performs three basic list operations: a) search, b) insert and c) delete on all three lists in a similar way to that described in Mattheakis and Papaefstathiou [2013]. The Portals buffer's portion, which is allocated to the PRQ, UMQ, and OFQ lists, is partitioned to equally sized segments, each one having fields for a *next pointer*, *initiator id* (message source), *size* and *match_bits* [Portals 2014] as well as a payload pointer to the memory. Figure 7 illustrates the 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. The unexpected message payload are copied into this Allocator Buffer, while the expected messages payloads are stored in the User Space and corresponding pointers are placed in the PRQ.

When the Portals initiator function is executed, the list manager allocates several OFQ entries, so that each unexpected incoming message can store its payload. In Figure 7, 128 (0-127) OFQ elements with 8,192 bytes each, are allocated, so that unexpected messages from different initiators can be stored in their own sub-buffer. The size of the OFQ entry is determined by the Portals *Init* function and depends on the application, while the match bits must be *0xff* to always match according to Portals match function [Portals 2014]. Every OFQ entry contains a '*local_offset*' field which shows the allocator buffer Section used for a specific unexpected message. Initially, the *local_offset* is zero. Whenever an unexpected messages arrives the *local_offset* advances by '*message_size*' bytes in order to be point to the space where the message payload can be stored. For this reason, a circular buffer is used in order to be able to re-use the same address spaces.

As the number of communicating nodes increases, the number of list entries increases accordingly [Keller and Graham 2010]. As a consequence, the time needed for a list search increases. In order to reduce this search time, Mattheakis and Papaefstathiou [2013] proposed a certain hashing scheme; this novel scheme has been utilized in our list manager and the required hash key is the source field of the Portals message. As the performance section clearly demonstrates, the use of hashing significantly increases the performance of PAC.

Fig. 8. Communication pattern for triggered rendezvous protocol.

3.2.5. Dynamic Memory Allocator. Our dynamic memory allocator implements the buddy dynamic memory allocation algorithm scheme [Knuth 1998], which can be very efficiently implemented in the hardware since it is comprised of mainly binary operations.

3.3. PAC Operations

3.3.1. Triggered Rendezvous Protocol and MPI-Functions. Our PAC system efficiently implements the rendezvous protocol as well as the one- and two-sided communications as described in Section 2; however, regarding the rendezvous protocol, we have altered it so as to reduce its intercommunication in certain cases. As described in Section 2, the eager intercommunication protocol ensures asynchronous processing in both expected and unexpected cases. However, in the case of unexpected messages, the specific Portals eager protocol either wastes network bandwidth due to the required payload retransmissions or uses significant amount of memory. In contrast, traditional rendezvous protocols (presented in Section 2.5) allow for asynchronous communication only in the case of unexpected messages, while the expected messages should be handled in a synchronous to the processing manner since the target CPU should be interrupted when a header arrives, so as to identify the PtlGet arguments.

Portals provides a mechanism through which an application can schedule certain message operations that are activated when a counting event reaches a value equal to a specified counting threshold. Those operations are called *triggered operations* [Barrett et al. 2011b] and can be efficiently utilized in collective offload commands [Schneider et al. 2013]. In this work, the *PtlTriggeredGet* operation is implemented by extending the *PtlGet* operation so as to be able to handle counting events and counting thresholds. Our augmented rendezvous protocol is implemented by utilizing those triggered operations so as to issue the target-side *get* request without involving the host processor as illustrated in Figure 8. In this example, the first *eager_limit* bytes of the message are sent to the target when the *PtlPut* operation is triggered. If the message is expected, the first part of the message is delivered directly into the target's UserSpace buffer; otherwise, it is delivered into the temporary buffers (OF list). A counting event that counts the bytes delivered is attached to the target buffer, and a *PtlTriggered-Get* is scheduled to be executed when a message larger than the *eager_limit* arrives. This approach works for both expected and unexpected messages, so the implemented novel rendezvous protocol (called triggered rendezvous protocol) provides asynchronous

communication in both cases. Figure 8 also presents the MPI send-receive non-blocking commands and how they utilize the Portals operations implementing this triggered rendezvous protocol. Specifically, the basic *MPI_IRecv* and *MPI_Isend* commands trigger the *PtlMEAppend*, *PtlTriggeredGet* and *PtlPut* operations respectively. In an expected message scenario, a *PtlTriggeredGet* operation is called which effectively posts, through the *PtlCTinc* function a *PtlGet* at the target node; otherwise (unexpected scenario), if the first part of an unexpected message arrives, a counting event is created and handled by the *PtlTriggeredGet*.

3.3.2. One-sided Operations and Global Arrays Functions. Our PAC module also efficiently implements one-sided operations as described in Daily et al. [2013]. In order to support such operations, the final memory destination of an incoming message should be determined at the target node by comparing certain contents of the message header with specific structures stored at the destination. In that respect, certain *match_bits* (first introduced in Section 3.2.4) are utilized. In particular, the *PtlWinCreate* command is implemented, which effectively inserts a one-sided priority list entry using unique *match_bits* to separate the different buffer requests, and then announces the *match_bits* to all one-sided participating nodes through a broadcast collective routine as described in Code I. In order to add this information, *PtlWinCreate* creates a *PtlWin*, which stores all the information about the one-sided participating nodes and their corresponding *match_bits*. In particular, this is a simple structure that contains *MatchBits* and *IntraNodes* arrays. An *IntraNodes* array has *NPROCS* elements of 1 bit which contain the nodes participating in a specific One-Sided communication, while the *MatchBits* array stores the *match_bits* of each node. As a result, each node gets the *match_bits* of the participating nodes by looking at the *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* from the *MatchBits* array. Subsequently, when the message arrives in node 2, only the one-sided cluster is traversed to find the appropriate PR entry and the appropriate buffer memory concurrently. In all cases, the target must trigger the execution of the *PtlWinCreate* operation before any one-sided messages arrives. Hence, a one-sided message is never unexpected since the corresponding PR entry is always inserted beforehand by the *PtlWinCreate* command. As a result, the UM and OF Lists are never traversed in one-sided communication.

The Global Arrays (GA) toolkit provides an efficient and portable “shared-memory” programming interface for distributed-memory computers. From the user’s perspective, a global array can be used as if it was stored in shared memory. In order to implement the Global Array protocol, we utilize our one-sided communication primitives.

CODE I. PtlWinCreate Function

Input: Uns32 size, Ptl_win win

Output: The base address of Pseudo Shared Memory (void * base_buf)

```

1 src = nodeId(); /* Get node ID */
2 if (base_buf != NULL){ /* Create PRQ entry with base_buf addr */
3   win->match_bits[src] = AssignUniqueMatchBit();
4   PtlPRInsert(base_buf, size, win);
5 }
6 else /* Set the node's match_bits equal to zero */
7   win->match_bits[src] = 0;
8 Broadcast(win->match_bits[src]); /* Send match_bits to other participant nodes */
```

Fig. 9. (a) GATable implementation (b) Global Array division with 9 elements.

In particular, a table (called *GATable*) is initially created through a *GA_Create* operation; the table contains `MAX_GA_ENTRIES`⁸ of *GAEntry* structure.⁹ Each *GAEntry*, in turn, contains the necessary Global Array characteristics, such as its name, type (integer, float, double), number of dimensions (*ndim*), the size of each dimension (*dims*), a pointer to the node's acceleration buffer, and one *PtlWin* entry as illustrated in Figure 9(a).

As specified in the GA protocol, the *GA_Create* command must be executed in all participating nodes in order to divide the Global array in equal sizes and distribute it in each node. For instance, in Figure 9(b), a *GA_Create* is called by four nodes and thus from the total 11 elements of the array, the first three nodes (0 up to 2) allocate three elements while node 3 allocates only the remaining two elements.

4. FROM ARCHITECTURE TO IMPLEMENTATION

In the development phase of our architecture, within the virtual high-level framework, 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. In order to implement such an entity into hardware, the exact interfaces to the other entities should be defined and the exact timing of the requested hardware system analytically described.

In order to see what kind of transformations are needed, Code-segment II compares the basic building blocks of our ListManager implementation in Untimed C (used in the architecture modeling phase) and Synthesizable SystemC (used for the hardware implementation) respectively. SystemC is selected because it is probably the most widely used System-Modeling language supported by all the major hardware design tools/vendors. In the process of converting the untimed high-level description to a cycle-accurate hardware implementation, the *clk* and *reset* signals must be added while all the input (*sc_in*) and output (*sc_out*) data analytically described. Our ListManager contains four different processes; one for list initialization and three for list operations (insert, search, delete). All processes are modeled as *SC_CTHREADS* [systemc 2014] as

⁸Implementation can support up to `MAX_GA_ENTRIES` different Global Arrays.

⁹Each *GAEntry* refers to one Global Array.

our High-Level Synthesis tool supports efficiently this type of description (share H/W resources, pipeline loops, etc.) [Cadence 2016].

CODE II. Untimed C Search Thread	SystemC Search Thread
<pre> 1 2 3 4 searchList(ListManager *list, 5 int listId, 6 SRAMEntry **current 7 ... 8){ 9 SRAMEntry *iter = 10 (list->allocated[listId]).head; 11 12 for(;iter;){ 13 <processing> 14 iter=iter->next; 15 } 16 }</pre>	<pre> sc_in<bool> clk, reset; sc_out<unsigned long>current; ... void listManager::search_thread(){ <Reset all output signals> current.write(0); wait(); while(1){ unsigned long iter = allocated[listId].head; for(;iter;){ <processing> iter = SRAM[iter].next; wait(); } }</pre>

The description of our ListManager's search function in a synthesizable SystemC thread is shown in code-segment II. The code before the first wait() resets all output 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 never returns, keeping the thread always alive. In contrast, it should call wait() to mark the end of a clock cycle and suspend the process until the next clock event. The significant difference between the Untimed C and SystemC is the iterator implementation; It is implemented as a simple unsigned long integer (pointers are not supported by most HLS tools) pointing to the next available SRAM position.

4.1. Disassociation of Data and Functionality

In certain cases, it is very important if the architecture and the functionality of a hardware module can be independent of the data type it supports. Higher-level languages like the ones used at the architectural description level (e.g., C, C++) include special keywords (e.g., void*) in order to disassociate the architecture from the data. The synthesizable subset 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 order to address this deficiency efficiently, we propose the following approach: The functionality of all the modules that are described using void pointers in the architectural description is encapsulated within our C++ synthesizable (i.e., SystemC) module and the architecture was separated from the data-dependent operations using C++ templates, as shown in the following code of the ListManager's Untimed C and Synthesizable C respectively.

In the case of Untimed C, the type of Portals queue is assigned dynamically through the init_data function. 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 entry.

CODE III. Untimed C Dynamic Memory Declaration

```

1 void initFastList(long SRAM_SIZE, long
2   LISTS_NUM,void (*init_data)(void*)) {
3
4   SRAMEntry SRAM[SRAM_SIZE];
5   for(i = 0;i < SRAM_SIZE;i++){
6     <Initialize the Data>
7     init_data(&(SRAM[i])->payload);
8   }
9
10  List allocated[LISTS_NUM];
11}

```

SystemC Static Memory Declaration

```

1 template<class DataType, long
2   SRAM_SIZE, long LISTS_NUM>
3
4   SC_MODULE(listManager) {
5     ...
6     SRAMEntry <DataType>
7       SRAM[SRAM_SIZE];
8     sc_in<DataType> element;
9     ...
10    List allocated[LISTS_NUM];
11}

```

4.2. Cycle-Accurate Timing Model

In the case of the untimed description, the functions modeling the functionality of an entity are fed with data whenever they are called. In a hardware implementation, each function is triggered at each clock cycle and thus certain signals should be added to mimic the architectural control flow. In this work, signals `enable_function_name` and `disable_function_name` are added at each function, so that a four-phase handshake protocol is applied by both the caller and the callee function threads. During the reset phase, both interface signals are low. Whenever the caller 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 completed, the callee checks whether the caller 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.

Code-segment IV shows the four-phase handshake protocol we utilize in the implementation of an example function (i.e., for MDTable traversing in the MD_Bind function); the wait statement inserted guarantees that each clock cycle performs one for iteration (representing the clock registers separating the combinational logic).

CODE IV. Cycle Accurate Timing Model MD_Bind

```

1 void PortalsAcc::MD_Bind() {
2   ...
3   while(1) {
4     while(enable_bind.read() == 0) {wait();}
5     disable_bind.write(1);
6     <function body>
7     for(i=0;i<MD_ENTRIES;i++) {
8       <processing>
9       wait();
10    }
11    ...
12    while(enable_bind.read() == 1) {wait();}
13    disable_bind.write(0);} }

```

4.3. HLS Tools' Optimizations

4.3.1. Loops Implementation. Typically, in designs described in HLS languages, combinational loops should be eliminated as they cannot be mapped to synchronous circuitry. A loop is combinational if an iteration of the infinite loop completes without executing a wait statement. In our implementation flow we used the CtoS HLS tool from Cadence [2011]; this tool allows the efficient implementation of loops by using two specific

approaches: (i) unroll and (ii) break. Loop unrolling eliminates the iterations by copying the code segment numerous times; in other words, all loop operations are executed in one cycle increasing dramatically the performance, together with the silicon area though. On the other hand, loop breaking is another option that simply inserts states in the loop and thus it increases the latency of the end system while it reduces the corresponding silicon cost.

CODE V. Higher Power of two

```
1 leadZeroPos = 32;
2 mask0 = 1 << (leadZeroPos-1);
3 for(;leadZeroPos>=0;leadZeroPos--){
4   if(number_input > mask0){ break;}
5   mask0 >>= 1;
6 }
```

Portals CT Table Allocation

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

In this work, both approaches are utilized. Specifically, if the loop contains a small number of operations and the total number of iterations is relatively small, then the unroll scheme is used, while in case of loops with many iterations the break approach are used so as to keep the critical path short. Code V demonstrates the synthesizable description of (a) the `higher_power_of_two` function which computes the higher power of two for a given 32-bit `number_input` and (b) the `CT_Alloc` function utilized in the CTable operation. In the `higher_power_of_two` loop, the unroll directive is used as the maximum number of iterations is 32 and the loop contains only one comparison and a shift left operation. On the other hand, in the CT Table loop, the break approach is used through a wait statement as the loop needs to read and write the quite large CTable array.

Loop unrolling proved a key optimization as it allowed us to explore a significantly bigger part of the hardware implementation solution space without modifying the SystemC specification: 14% higher frequency was obtained with an area overhead of 6%.

4.3.2. Memory Implementation. Another important implementation characteristic has to do with the mapping of an array to physical memory. CtoS supports four possible options for this mapping: (i) *flatten*, (ii) *built-in*, (iii) *prototype*, and (iv) *vendor* array. Each option is optimal for a different array use case and/or access pattern. The *flatten* option creates an array of registers; this option is only suitable for very small arrays as it increases the silicon area significantly. The *built-in* operation implements the arrays as built-in SRAMs; this is the desired choice when the size of the array, in words, is medium (up to 256 words or fewer [Cadence 2016]), and multi-process access is required. The *prototype* option does not correspond to any actual hardware and it is intended only for design exploration; it should be replaced if the code should be synthesized. Finally, the *vendor* memory is treated as a black-box and it is ignored in the scheduling and synthesis processes. The designer should implement a *vendor* memory using any technology cell or any synthesizable modeling technique; this is typically a better option for large arrays. In this work, the *built-in* and *vendor* options are used as the flatten option incurs a significant area penalty. More specifically, the Accelerator Buffer (Figure 6) is implemented with fast (2 read ports/2 write ports) dedicated SRAMs (which are totally independent from the CPU cache and/or main memory) using the *built-in* option as the accesses to this memory should be performed very fast since they are part of the critical path. The buffer for unexpected messages payload is implemented using a 4MB Vendor DRAM; this DRAM is described in handwritten Verilog Vendor DRAM, the Vendor DRAM technology library is used to describe the DRAM constraints and an xml file is used to integrate the above DRAM with the rest of the design.

Fig. 10. Hardware accelerator verification phases.

Table I. NAS Benchmark Suite

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

4.4. Hardware Acceleration Verification

One of the main advantages of our novel design approach is that it allows for a common verification environment that spans across the different abstraction levels of the SoC design [Cadence 2013]; our system is verified along three stages as illustrated in Figure 10. At all stages, the same actual testing software suite is used. In the first stage, the synthesizable SystemC simulation results are compared (Figure 10(a)) with those of the instruction-accurate virtual prototype design. In the next step (Figure 10(b)), the Verilog behavioral model generated by CtoS is simulated and its outputs compared with those of the previous step. Finally, in the third verification step (Figure 10(c)), the output of the synthesizable RTL Verilog together with the exported RAMs, as generated by CtoS, is compared with that of the 2nd step. In order to support all those verification steps, certain verification wrappers were developed so as to identify the potential simulation mismatches.

5. PERFORMANCE EVALUATION

5.1. Benchmarks

Four representative benchmarks of the NAS Parallel Benchmark (NPB) suite Table I and one molecular dynamic benchmark were used in order to evaluate our novel PAC system.

The NPB comprises of a set of parallel applications designed to evaluate the performance of supercomputers [Bailey et al. 1994]. The benchmarks are derived from computational fluid dynamics (CFD) applications, adaptive mesh, parallel I/O, multi-zone applications, computational grids, and the like. We select IS, FT, EP, and DT as they utilize a wide range of MPI routines using problem size ‘S’, while our implementation of the MPI collective commands are based on the openMPI implementation [OpenMPI 2016]. EP is an “embarrassingly parallel” kernel, which evaluates an integral by performing pseudorandom trials. FT is a 3D partial differential equation solver using FFT. IS implements a parallel integer sorting algorithm. Finally, DT (Data Traffic) works with randomly generated data using quad-trees (black hole and white hole) and binary shuffle as task graphs [Frumkin 2005].

Fig. 11. Performance of Intel E8400, ARM A9, and hardware Portals processor.

The MD benchmark is used to simulate multi-particle systems ranging from biomolecules on Earth, to the motion of stars and galaxies in the Universe. We select the MD Lennard-Jones benchmark which computes energy fluctuation per particle in a wide range of pressure and temperature values [Hernández 2008] since this utilizes the GA approach.

5.2. Worst Case Performance

In this section, a benchmark presented in Brightwell and Underwood [2004] is used to evaluate the efficiency of our novel PAC system when handling the unexpected queue which is the slowest module in our system; in that benchmark, a queue with a predefined number of entries is handled under the worst-case scenario which is that a node has received numerous unexpected messages and the matching entry is always found at the queue's tail. [Flajslik et al. 2016] have accelerated the tag-matching task, in software, significantly. Our work though is more efficient than this approach since it is based on the work in Mattheakis and Papaefstathiou [2013], which outperforms their optimized tag matching performance. In contrast to Mattheakis and Papaefstathiou [2013], in our benchmark, all nodes send messages to node 0 and their tags are in descending order, while node 0 receives messages with tags in ascending order, and as a result the application is forced to always match the last element in the UMQ.

Figure 11 shows our embedded portals software, Portals processor in the hardware scheme and a high-end host processor executing the above realistic scenario. Results show that our Portals hardware processor (with no hashing) is at least 1 order of magnitude faster than both the high-end processor and the embedded CPU (with no hashing). Initially, high-end CPU processor is faster than embedded Portals software, but after 16,384 messages high-end processor suffers from performance degradation due to a high number of cache misses. On the other side, for the simulated embedded processor, we use 1MB cache, while in Portals hardware we use the same size of cache with two ports for read/write for the sake of simulation speed.

Moreover, 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 two orders of magnitude faster than the embedded CPU. Finally,

Fig. 12. Performance of ARM A9 and hardware Portals processor in shorter queue lengths.

Fig. 13. Speedup of NAS parallel benchmarks.

Figure 12 illustrates the processing time of our PAC and ARM A9 when no hashing is employed for queue lengths equal to 1, 2, 4, 8, 16.

5.3. Typical Performance

In order to measure the typical performance of our system, we have used the five representative benchmarks described in Section 5.1.

Figure 13 demonstrates the speedup triggered by (a) the software-based approach utilizing our hashing function, (b) our PAC module when hashing is de-activated, and (c) our PAC system with hashing activated for all benchmarks, within a 32-node platform (left) and a 128-node platform (right); the reference implementation is the

standard portals library executed on an ARM A9. The speedup of the S/W hash-based approach varies from 10% to over 30%; this speedup is much lower to that reported by Mattheakis and Papaefstathiou [2013] for the relevant intercommunication routines, since in our case we measure the total intercommunication time and not just the queue processing part. The speedup triggered by our novel PAC system varies from 50x, when no hashing is applied to over 130x when we activate our hashing scheme within a 32-node Platform.

The reason that the PAC speedup is so much higher when hashing is activated is mainly the following. All our HW modules except the ListManager have a constant latency irrespective of the input data; as a result the critical module is the ListManager. So our hashing scheme accelerates significantly the list management tasks, especially in the case of collective intercommunication routines which traverse a significant number of queues, such as MPI_Reduce, MPI_AllReduce, and MPI_AlltoAll. On the other hand, in one-sided communications (i.e., MD benchmark) our hashing scheme does not accelerate the overall intercommunication processing, as the source node is not known to the GA_Create routine. It is important to highlight that when moving to larger systems, the speedup grows further, from 80x to over 200x as demonstrated in Figure 13(b). Moreover, if we adopt the performance metric presented in Brightwell and Underwood [2004], the speedup triggered by PAC, when utilized in multi-thousand node systems, is expected to be even higher than that of Figure 13(b).

5.4. Comparison with Another Hardware-Based Accelerator

In this section, we evaluate the PAC performance compared with the only known hardware-based accelerator of an intercommunication protocol – the MPI accelerator Mattheakis and Papaefstathiou [2013].

5.4.1. Portals Accelerator versus MPI Accelerator. Portals is a network programming interface which can support not only point-to-point interfaces, such as MPI, but also various partitioned global address space models such as PGAS. Hence, the Portals Accelerator implements additional functionality compared to the MPI Accelerator described in Mattheakis and Papaefstathiou [2013]. Thus, we measure the performance overhead of Portals Accelerator when the following procedures are executed in the two hardware systems¹⁰: PtlMDBind, PtlMDRelease, PtlCTAlloc, PtlCTFree, OF_Insert (MPI_Init), and OF_Search.

The main differences of the two hardware systems are the following: The PtlMDBind and MD_Release Portals functions traverse the complete MD_Table so as to find the Memory Region in the initiator's User Space, which has been allocated before start sending its data, while MPI allocates the User Space during the MPI_Send routine. Moreover, Portals uses counting events to acknowledge the transaction completion when traversing the CT_Table in the PtlCTAlloc and PtlCTFree procedures, while the MPI Accelerator simply waits to receive the Data Packet in the corresponding MPI_Wait procedure. Finally, PAC uses an overflow list to save the payload of the Unexpected Messages, while the MPI Accelerator allocates space for UM's payloads during the execution of the MPI Send routine.

In order to see what the overhead of the portals' middleware is, we first compare the performance when plain MPI and MPI over Portals are executed in an ARM A9 CPU with and without hashing, for our four MPI-based NPB benchmarks. Figure 14 demonstrates that this overhead is from 2% to about 23% when no hashing is employed which is reduced from about 1% to approximately 11% when software hashing is activated.

¹⁰These routines trigger the additional portal's overhead when compared with the monolithic MPI corresponding routines.

Fig. 14. Portals-MPI Accelerator Communication Overhead using NAS Benchmarks in 128node.

Fig. 15. Performance of Eager and Rendezvous Protocols.

Moving to the comparison of the hardware systems we see that PAC is from 1% to less than 20% slower than the MPI hardware accelerator while it provides the additional functionality and flexibility of the Portals middleware. In more detail, the benchmarks that handle more Unexpected Messages and thus the Overflow List must be traversed more frequently (i.e., EP and FT) trigger the higher overhead.

5.5. Rendezvous versus Portals Eager Protocol

In this section, we measure the performance of the Portals Eager Protocol as well as the Triggered Rendezvous Protocol, as described in Section 3. In Figure 15, we use the MPI AlltoAll Routine and each Data Packet (DP) is 8KB. In our reference implementation, we use 1MB of dedicated SRAM per PAC for all the messages under processing. As a result, the maximum message our PAC can handle can be up to this size (i.e., in the case that there is only a single message received by a single node); obviously the more the messages received simultaneously the smaller they should be so as to fit into

Fig. 16. Power and area estimation flow.

our dedicated memory. Our architecture is fully scalable so the size of the SRAM can arbitrary grow if there is such a need (at the cost of the additional silicon).

We measure the performance of the Portals Eager Protocol and of two versions of the Triggered Rendezvous: (i) with eager limit of 0 bytes, and (ii) with eager limit of 4KB. In the Portals eager protocol, the sender simply sends the whole DP, while in the Triggered Rendezvous with an eager limit equal to zero the sender sends only the header of the DP to the receiver, and the receiver issues a *get* command to retrieve the message. Finally, in the Triggered Rendezvous case with eager limit equal to 4KB, the sender sends the header of the DP and 4KB of the payload to the receiver, which should then perform a *get* the remaining 4KB. In other words, in the Portals Eager Protocol there is one transaction (Sender sends the whole DP), while in the Triggered Rendezvous transactions, there are three transactions: (i) the sender sends the header and maybe a part of the DP, (ii) the receiver issues a *get* to retrieve the DP, and (iii) the sender sends the remainder of the DP.

In the Portals Eager Protocol, there is a single transaction, but the message might be unexpected; if the message is unexpected and it contains a payload, both UM & OF lists should be traversed in order to save the header and payload, respectively, while in the case of unexpected messages without payloads only the UM list is traversed so as to save the header of the message. As a result, and as Figure 15 clearly demonstrates, the most efficient option is the Triggered Rendezvous with an Eager Limit of zero since it is the only one that cannot trigger any unexpected messages that contain any kind of payload and thus it will never have to traverse the large OF list, while the time needed to copy the message to the Message Buffer and then to the User Space is eliminated. This also explains why the Portals Eager Protocol has the worst performance; it requires more time in order to copy 8KB first to the Message Buffer and then to the User Space (in case of unexpected messages) than the 4K Triggered Rendezvous which handles 4K data items.

5.6. Area and Power Results

In this section, we evaluate the Area and Power consumption of PAC as a whole as well as its components (ListManagers, Allocator etc). The silicon results were obtained using the Cadence C-to-Silicon high-level synthesis tool with the 65nm Europractice TSMC standard-cell library. Power results are reported in Typical Conditions (TC). However, our PAC being designed in fully Synthesizable System-C; as a result it can also seamlessly be implemented in any reconfigurable platforms.

5.6.1. Area & Power of H/W Accelerator. The area and power results were produced using Cadence's Incisive Enterprise Simulator (INCISIV) and Encounter RTL Compiler (RC) as described in Figure 16, while we measures the SRAM Dynamic Power results using CACTI 6.5 [HP 2009]. We use the most demanding routine (i.e. MPI AlltoAll) as the testbench in a 128-node platform. Table II summarizes the Portals Accelerator's SRAMs and shows the Dynamic Power Consumption of each one as computed by the CACTI model.

Table II. Dynamic Power Consumption for SRAMs

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
Memory Counting Event Table	512 x 128 bits	4 mW/GHz

Fig. 17. Silicon results using MPI_AlltoAll benchmark.

Figure 17(a) illustrates PAC's Power consumption as well as SRAMs' Dynamic Power consumption as reported by the Cadence and CACTI models, respectively, while Figure 17(b) illustrates the silicon area of only the processing part of PAC (i.e., without the silicon for the SRAMs) for different targeted frequencies. Both Power and Area grow linearly with the frequency, while they are increased only by 2.2x and 1.39x, respectively, when tripling the targeted frequency. Looking to the corresponding numbers for the different implementations of an ARM Cortex A9 [ARM 2013], our system has at least two orders of magnitude lower power consumption and at least an order of magnitude lower silicon cost.

6. CONCLUSIONS

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 the high intercommunication delay. Although there are several approaches trying to hide this 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 thousands nodes. This is a time-consuming task, hence offloading it from the main processor, has emerged as an efficient way to reduce the intercommunication delay. At the same time, the Portals intermediate communication protocol is a very promising approach for parallel systems' intercommunication since it efficiently supports both point-to-point communication interfaces (such as MPI) as well as various partitioned global address space (PGAS) intercommunication models. Therefore, any acceleration of the Portals protocol triggers significant speedups for both MPI and PGAS schemes.

In this article, we present PAC which is the first known SoC acceleration system for the Portals' protocol. We also describe a novel modeling, design and implementation approach which can accelerate significantly the development of any CPU accelerator. Our experimental results shows that our novel system is from one and up to three orders of magnitude faster than two general-purpose CPUs executing this same protocol, with approximately 15% time overhead when compared with a hand-made purely MPI H/W Accelerator. Moreover, PAC is up to 2 orders of magnitude faster than an ARM A9 in both MPI and GA benchmarks while, and more importantly the speedup is grown with the number of nodes in the parallel system. Additionally, our accelerator consumes approximately 100 times less power and it is being implemented at 1/10th of the silicon area of a small embedded CPU.

In the future, we plan to evaluate the performance of our PAC with larger, real-life parallel applications. Additionally, we will provide the PAC source code and the hardware development flow in an open-source manner to academia.

ACKNOWLEDGMENTS

The authors would like to thank Dimitrios S. Nikolopoulos for his very constructive suggestions.

REFERENCES

- ARM 2013. <http://www.arm.com/products/processors/cortex-a/cortex-a9.php>.
- D. Bailey, E. Barszcz, J. Barton, and D. Browning. 1994. *The NAS Parallel Benchmarks*. RNR Tech. Rep., Mar. 1994.
- B. W. Barrett, R. Brightwell, K. S. Hemmert, K. Pedretti, K. Wheeler, and K. D. Underwood. 2011a. Enhanced support for openshmem communication in portals. In *Proceedings of the IEEE 19th Annual Symposium on High Performance Interconnects (HOTI)*, Aug. 2011, 61–69.
- Brian W. Barrett, Ron Brightwell, K. Scott Hemmert, Kyle B. Wheeler, and Keith D. Underwood. 2011b. Using triggered operations to offload rendezvous messages. *EuroMPI 2011*. 120–129.
- Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. 2014. The Portals 4.0.2 networking programming interface. Tech. Rep. SAND2014–19568. Sandia National Laboratories.
- Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzone, William Harrod, Jon Hiller, Sherman Karp, et al. 2008. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. University of Notre Dame CSE Department Technical Report. TR-2008–13, Tech. Rep, Sept. 2008.
- S. Borkar and A. A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5, 67–77.
- R. Brightwell and K. D. Underwood. 2004. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 9.
- R. Brightwell, R. Riesen, and K. D. Underwood. 2005. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl* 2, 103–117.
- Ron Brightwell, Kevin T. Pedretti, Keith D. Underwood, and Trammell Hudson. 2006. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro* 26, 3, 41–57.
- CADENCE 2013. <http://www.eejournal.com/archives/articles/20131031-cadence>.
- CADENCE 2016. <http://www.cadence.com>.
- Cadence C-to-Silicon Compiler User Guide. 2011. Product Version 11.10 s200, August 2011.
- L. Cai and D. Gajski. 2003. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*. ACM, New York, 19–24.
- Sung-eun Choi, Evan Harvey, Howard Pritchard, James Shimek, James Swaro, and Zachary Tiffany. 2016. The GNI provider layer for OFI libfabric. In *Cray User Group Conference, May 2016, London*.
- Luc Claesen, Maria-Teresa Sanz-Pascual, Ricardo Reis, and Arturo Sarmiento-Reyes. 2015. VLSI-SoC: Internet of things foundations. In *Proceedings of the 22nd IFIP WG 10.5/IEEE International Conference on Very Large Scale Integration*, Oct. 2014.

- Jeffrey Daily, Abhinav Vishnu, Bruce Palmer, and Hubertus Van Dam. 2013. PGAS models using an MPI runtime: Design alternatives and performance evaluation. *SCHEDULE*. Nov. 2013. <http://sc13.supercomputing.org/>.
- Said Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter. 2015. ATOS. The BXI interconnect architecture. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, 18–25.
- M. Flajslik, J. Dinan, and K. D. Underwood. 2016. Mitigating MPI message matching misery. In *Proceedings of the International Supercomputing Conference*.
- Michael Frumkin. 2005. Data flow pattern analysis of scientific applications. In *Proceedings of the Workshop on Patterns in High Performance Computing*.
- Eduardo R. Hernández. 2008. Molecular dynamics: From basic techniques to applications. In *Institut de Ciència de Materials de Barcelona, AIP Conference Proceedings*. 1077. 1, Nov. 2008.
- HP 2009. <http://www.hpl.hp.com/research/cacti>.
- R. Keller and R. L. Graham. 2010. Characteristics of the unexpected message queue of MPI applications. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*. 179–188.
- D. E. Knuth. 1998. *The Art of Computer Programming, 1 Fundamental Algorithms*. 3rd ed: Addison-Wesley Longman Publishing Co, Inc, Boston, MA.
- Pavlos M. Mattheakis and Ioannis Papaefstathiou. 2013. Significantly reducing MPI intercommunication latency and power overhead in both embedded and HPC systems. *ACM Trans. Archit. Code Optim.* 9, 4, Article 51 (January 2013), 25 pages.
- OpenMPI 2016. <https://www.open-mpi.org/>.
- OVP 2014. <http://www.ovpworld.org>.
- PGAS 2013. http://en.wikipedia.org/wiki/Partitioned_global_address_space.
- SystemC 2014. <http://www.asic-world.com/systemc/process3.html>.
- Timo Schneider, Torsten Hoefer, Ryan E. Grant, Brian W. Barrett, and Ron Brightwell. 2013. Protocols for fully offloaded collective operations on accelerated network adapters. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*. IEEE, 593–602.
- UCX 2016. <http://www.openucx.org/>.

Received May 2016; revised September 2016; accepted October 2016