Significantly Reducing MPI Intercommunication Latency and Power Overhead in Both Embedded and HPC Systems

PAVLOS M. MATTHEAKIS, Telecommunication Systems Institute, Technical University of Crete,
and University of Crete
and IOANNIS PAPAEFSTATHIOU, Synelixis Solutions Ltd

Highly parallel systems are becoming mainstream in a wide range of sectors ranging from their traditional stronghold high-performance computing, to data centers and even embedded systems. However, despite the quantum leaps of improvements in cost and performance of individual components over the last decade (e.g., processor speeds, memory/interconnection bandwidth, etc.), system manufacturers are still struggling to deliver low-latency, highly scalable solutions. One of the main reasons is that the intercommunication latency grows significantly with the number of processor nodes. This article presents a novel way to reduce this intercommunication delay by implementing, in custom hardware, certain communication tasks. In particular, the proposed novel device implements the two most widely used procedures of the most popular communication protocol in parallel systems the Message Passing Interface (MPI). Our novel approach has initially been simulated within a pioneering parallel systems simulation framework and then synthesized directly from a high-level description language (i.e., SystemC) using a state-of-the-art synthesis tool. To the best of our knowledge, this is the first article presenting the complete hardware implementation of such a system. The proposed novel approach triggers a speedup from one to four orders of magnitude when compared with conventional software-based solutions and from one to three orders of magnitude when compared with a sophisticated software-based approach. Moreover, the performance of our system is from one to two orders of magnitude higher than the simulated performance of a similar but, relatively simpler hardware architecture; at the same time the power consumption of our device is about two orders of magnitude lower than that of a low-power CPU when executing the exact same intercommunication tasks.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; D [Software]: Concurrent Programming, Parallel Programming

General Terms: Design, Performance

Additional Key Words and Phrases: MPI, HPC, memory management, architectural exploration, intercommunication, embedded systems

ACM Reference Format:

Mattheakis, P. M. and Papaefstathiou, I. 2013. Significantly reducing mpi intercommunication latency and power overhead in both embedded and hpc systems. *ACM Trans. Architect. Code Optim.* 9, 4, Article 51 (January 2013), 25 pages.

DOI = 10.1145/2400682.2400710 <http://doi.acm.org/10.1145/2400682.2400710>

1. INTRODUCTION AND MOTIVATION

In the last few decades the silicon industry has enjoyed significant density, speed, and power improvements every year. However, in the next years only density will continue

Authors' addresses: P. M. Mattheakis (corresponding author), Telecommunication Systems Institute, Technical University of Crete, Kounoupidiana, GR73100, Chania, Greece and Computer Science Department, University of Crete, Heraklion, Greece; email: pmat@csd.uoc.gr; I. Papaefstathiou, Synelixis Solutions Ltd., 10 Farmakidou St., Chalkida, GR34100, Greece.

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.

© 2013 ACM 1544-3566/2013/01-ART51 \$15.00

DOI 10.1145/2400682.2400710 <http://doi.acm.org/10.1145/2400682.2400710>

its previous growth, as reducing the size of a transistor will have less impact on its speed and its power consumption than in the past. Systems with numerous homogeneous and/or heterogeneous 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 [Borkar and Chien 2011].

Highly parallel systems are comprised of a vast number of cores. The combined processing power in such systems is just an indication of system's performance, as many factors can trigger significant system's underutilization. One such key factor is communication, during which cores may remain idle, waiting to synchronize or to exchange data with each other. Increasing the number of cores in a system also increases the percentage of total execution time spent on communication, as each core has less work to do and usually more other cores to communicate with. Hence, reducing communication's overhead is a crucial challenge as the core count increases.

One of the most popular communication protocols in parallel systems is the Message Passing Interface (MPI). MPI standard [Forum 1994] includes point-to-point and collective routines and was introduced in the early 90s as a mean of programming distributed memory parallel systems. One of the most significant goals of all the MPI implementations has been to minimize the MPI overhead, i.e. the time spent by a processing node executing the MPI-related tasks.

Several approaches have been introduced in the past in order to minimize this overhead. For example, nonblocking point-to-point operations allow the programmer to overlap computation with communication thus eliminating CPU idle time. However, aside from the difficulty to extract code which can be overlapped with communication, nonblocking operations also suffer from a number of side-effects, such as the growth of MPI unexpected message and posted receive queues, namely UMQ and PRQ respectively, which, in turn, significantly affects the MPI overhead as outlined in the next paragraph and further demonstrated by our experimental results.

UMQ grows with programs which use eager nonblocking send commands and in which the sender assumes that the receiver has enough space to buffer the messages sent. PRQ, on the other side, grows with asynchronous receive commands. The size of both queues affects MPI's communication overhead [Underwood and Brightwell 2004], as applications may traverse a significant number of entries searching for a certain message. Increasing the number of processes in order to get more parallelism usually translates to smaller messages and hence extensive usage of the eager protocol [Brightwell and Underwood 2004; Brightwell et al. 2005a]; for numerous applications, the above translates to longer UMQs and PRQs [Brightwell et al. 2006; Keller and Graham 2010].

The size of the MPI queues is one of the major factors which determine messages' latency, independently from the underlying MPI implementation [Underwood and Brightwell 2004]. Hence, accelerating queue operations can result in reduced MPI overhead [Brightwell et al. 2010]. The above observation has led to MPI implementations in which the MPI queue handling is offloaded to embedded processors, to Network Interface Cards (NICs) or even to dedicated hardware.

The contributions of this article are twofold. First we present an existing 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 article presents certain novel MPI acceleration cores that have significant advantages over the existing approaches. Firstly, they can execute the most frequently used MPI operations orders of magnitude faster than the existing CPU-based approaches; this results in a significant reduction of the execution time of several high-end parallel

Fig. 1. Nonblocking receive operation.

applications. At the same time their small silicon footprint allows for their efficient use in embedded parallel systems. It should be stressed that even though until recently MPI was considered prohibitively expensive (in terms of processing overhead) for embedded systems, this assumption is rapidly changing. There are already embedded systems prototypes supporting MPI [Go andhringer et al. 2010]; the designers of those systems have identified that MPI is a heavy protocol when executed on low-power CPUs. So another significant advantage of our approach is that it can extend the use of MPI to the next-generation highly parallel embedded systems.

The rest of the article is organized as follows. Section 2 introduces MPI's nonblocking operations and MPI's asynchronous communication primitives. Section 3 presents a framework allowing for the rapid exploration of novel parallel architectures when executing real-world applications. In Sections 4 and 5, two architectures for offloading certain MPI tasks are evaluated on top of the presented framework, when executing certain widely used parallel benchmarks. Section 6 describes in detail the implementation of our cores while Section 7 presents our real-world experimental results, based on different parallel benchmarks. Finally, Section 8 outlines the existing work in the field and highlights the advantages of our approach while Section 9 concludes this article and gives certain directions for future research.

2. BACKGROUND

This section initially introduces the concepts of nonblocking operations as that of asynchronous communication primitives. Then, it presents the infrastructure needed in order to actually translate the former to the latter.

Figure 1 shows an abstract view of a two-processor distributed memory system when exchanging an MPI message. Processor $P1$ executes a point-to-point MPI send command and processor $P2$ executes a matching MPI receive command so as to accept message MSG . The MPI send command is blocking which means that $P1$ is not able to resume its computation unless MSG is dispatched. $P2$ on the other side either executes a blocking receive operation, or follows a nonblocking receive scenario.

In the first blocking case, which is also depicted in Figure 2(a), if the message has not yet arrived when $P2$ executes the blocking receive command, $P2$ would remain in a blocked state until the actual message finally arrives. Otherwise, if the message has already reached $P2$ when the receive command is executed, it would be delivered instantly to the processor and the computation would be resumed. Nonblocking MPI commands aim at reducing the idle time of their blocking counterparts by injecting a portion of the computation following the blocking command, which does not depend on the command issued, right after the nonblocking command itself (just like the delay slots in pipeline processor but at a greater code granularity). Incorporating such a nonblocking scheme in the presented message exchange scenario results in the command sequence shown in Figure 2(b). The nonblocking receive returns instantly

Fig. 2. MPI receive schemes.

Fig. 3. MPI nonblocking receive schemes.

without waiting for the message, and the processor resumes its computation. At some point, P_2 needs the data contained in the message and thus waits for the message to arrive by executing an MPI *wait* command. As soon as the message arrives, the MPI wait command returns indicating the successful arrival of data and the computation is resumed. The parts of the computation which are executed before and after the arrival of the message are denoted as *Computation_o* and *Computation_d*, respectively.

The preceding description is a very simplified one and as a result further analysis should be performed in order to accurately predict system's performance. Figure 3(a) describes in more detail the operations executed by P_2 during a nonblocking receive. In the abstract view it was assumed that a nonblocking 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 the receive command. One of the most significant such operations is the search in the unexpected message queue in order to check whether a send command matching the specified receive has been issued in the past ("Process PR" in Figure 3(a)). Accordingly, when *MSG* arrives in P_2 the computation is not instantly resumed, since the posted receive queue has to be traversed in order to check whether a matching receive has been posted in the past ("Process UM" in Figure 3(a)). Both of the aforesaid linear searches have to be performed by the host processor, incurring a significant overhead in the case of large queues.

Offloading the queue processing from the host processor to an accelerator allows for extensive overlap between the actual computation and the communication tasks, as the corresponding accelerator operates on the queues at the same time that the host processor executes the rest of the application. This fact is shown in Figure 3(b), where an accelerator accesses the PRQ and UMQ allowing the host processor to continue its computation. Hence, by introducing offloading schemes in the non-blocking scenario, more time is granted for the host processor, further increasing the asynchronicity in the interprocess communication. Numerous such offloading schemes have been introduced in the past in either academic works or industrial products, as

detailed in the related work section; their experimental results clearly demonstrate that offloading portions of the MPI stack significantly reduces the communication overhead while increasing the overall performance of the system.

In order to evaluate such an MPI offloading system one needs to be able to create a full parallel distributed memory system executing a complete, real-world if possible, parallel application. Until now this was mainly possible when the actual silicon of the MPI device was implemented; at the design phase only rough simulations based on small synthetic benchmarks were utilized.

In this article, we propose a novel framework for exploring-accelerator centric architectures. This framework is based on a virtual platform environment, emulating all the popular processor architectures whilst also allowing for accelerator modeling in high-level programming languages. In this way, the proposed architectures can be evaluated with real-life software, running on the emulated processors and not only with synthetic benchmarks just triggering corner cases. The introduced platform is supported by a novel rapid flow to silicon based on a high-level synthesis industrial-scale EDA suite. The above flow requires translating the hardware models to synthesizable code. As this process is not trivial, directions are given which guarantee implementations of acceptable quality. The beforementioned platform is used as a vehicle to quantify two accelerator-based architectures for offloading a portion of the MPI stack to dedicated hardware. The first architecture is based on a list manager scheme, similar, yet more sophisticated and thus more efficient, to the one presented in Underwood et al. [2005b]. The second architecture introduces a simple hashing scheme effectively reducing the search time with negligible impact to the insert and delete times. The previous architectures are evaluated with several benchmarks utilizing nonblocking receive commands. Both architectures exhibit computation-communication overlap, with the second one allowing for even more space for computation. Finally, following the introduced path to silicon, both architectures are quantified with regards to area, power, and performance metrics.

The differentiation of the proposed work when compared with the existing as well as already proposed similar systems can be summarized in the following points.

- The proposed architectures are integrated in a platform consisting of state-of-the-art CPUs executing real-world parallel software.
- A path to actual implementation is presented, accurately quantifying the proposed architectures with respect to area, power, and performance.
- This is, to the best of our knowledge, the first such MPI offloading system that has been fully synthesized and thus the area, power, and performance of the actual silicon is presented; the related work in the area, of hardware offloading systems either presents the high-level architecture of such a system without giving any discrete results for performance, area, and power or implements just a single MPI command (different from the ones we implement) in a single FPGA without incorporating it within a real parallel system.

3. ARCHITECTURE EXPLORATION FRAMEWORK

This section presents a framework for exploring accelerators' architectures within a highly parallel system while mainly focusing at the exploration of our novel MPI offloading engine. The presented framework is based on a virtual platform simulation environment "optimized for SoC and MPSoC platforms". In order to prove the accuracy of the introduced framework, a certain benchmark of the widely used NPB suite is used. Results indicate that the introduced framework fully succeeds in capturing the behavior of MPI's internal structures, when compared with the results measured from actual off-the-shelf parallel systems executing the exact same MPI code.

3.1. OVP

The development of each new embedded platform is mainly comprised of two development phases, the hardware and the software one. With the former phase lasting several months and the very aggressive product schedules, the latter phase should be initiated as early as possible. The most common practice is to start the software development in a host machine running a general-purpose operating system before the real hardware or its prototype are functional. When the prototype is available, developers port the software using cross compilers and relevant tools. Later on, when the final hardware is available, further modifications are needed for the product release.

The aforesaid embedded product development procedure faces many challenges. First of all, the development environment of a host machine is significantly different than the one of the target hardware. A modern-day example is the software development for MPSoCs where separate processors are emulated by distinct host threads. This approach provides limited “debugability” when tracking down complex multiprocessor bugs and issues, e.g., race conditions.

The focus of Open Virtual Platforms (OVP) [OVP 2012] is to reduce the product development time, especially for SoC and MPSoC platforms. OVP allows instantiating processor models for almost all popular architectures (e.g., ARM, PowerPC, Sparc, etc.) along with models for widely used peripherals (e.g., USB host controllers, DMA engines etc.). The under development hardware modules can be co-simulated with the above processors and peripherals, by creating a corresponding high-level model description; this description should be abstract enough so as to allow for an acceptable simulation speed but at the same time accurate enough so as to expose the most significant parts of the underlying architecture and the way that the architectural features affect the system’s behavior. This approach allows software engineers to write software for the target hardware platform (including the accelerator) at the very early stages of the product development.

Accurate high-level descriptions of complex systems would have been useless, without the support of an efficient simulator. Hence, Imperas Inc. [Imperas 2012] which initiated the OVP project, has made available the OVPsim simulation tool which allows for executing large-scale software on complex MPSoC platforms. OVPsim is able to model multiprocessor systems with shared or distributed memory models, caches, buses in arbitrary topologies and peripheral models. Besides expressiveness, OVPsim is also very fast when simulating large-scale systems. Its performance depends on many factors (e.g., processor scheduling interval), but a typical estimation is hundreds of millions of simulated instructions per second. With respect to debugging capabilities, OVPsim provides hooks to almost all popular external debuggers (e.g., GDB).

3.2. Virtual Platform

One of the aims of this work is to provide a framework for exploring accelerator’s architectures especially for offloading the MPI processing from the CPUs. Our framework is based on the OVPsim; an overview of this platform is shown in Figure 4. The platform consists of N processing nodes and each node is, in turn, comprised of a processor model, a memory module, a bus, a dynamic memory allocator peripheral, and an MPI accelerator.

The Open Risc 1000 (OR1K) [Open Risc 2012] processor model was selected to be the computation core of each node; however, the exact same design process can be followed in the case any other processor is selected (like the ARM or an Intel one), while the presented results have proved to be processor independent. OR1K’s model can be configured during instantiation by the OVPsim’s Innovative CPU Manager (ICM) API. For example, OR1K can be configured with either a 32-bit or a 64-bit logic

Fig. 4. Platform.

address space. In addition, certain functions e.g., `printf`, can be intercepted, so that the I/O is performed at the host machine and not at the simulated platform. The aforesaid feature along with the ability to hook the execution software running on the processor directly and seamlessly to the GDB debugger significantly accelerated the development process of the multiprocessor system.

During system's simulation, the software allocated to each of the OR1K processors is executed in the host machine at predefined time intervals. In particular, a certain number of instructions which correspond to the amount of time allocated to each processor are executed, then the scheduler stores the processor state and triggers the execution of the software assigned to another processor for the same predefined interval. This pseudo-parallel approach emulates the concurrent behavior of the actual system. For the simulations performed in this work, a timing interval of 1 ms was selected which gave a good trade-off between simulation time and accuracy of the results; for the latter there is a analytical description in the next subsections.

The memory of each node consists of two parts, instructions and stack. Those parts, along with the space reserved for the memory mapped intercommunication with our MPI offloading engine, are shown in Table I. The field which corresponds to the MPI accelerators is consistent among all the processors. This means that there is a unique global addressing scheme which associates an address interval with the MPI buffer of a specific node. Hence, sending a message with sender $node_i$ and receiver $node_j$ is implemented as a memory write at address $0x60000000 + j * N * MAX_MESSAGE_SIZE + i * max_MESSAGE_SIZE$ which

Table I. Memory Mapping

Address		Mapped
Low	High	
0x00000000	0x0ffffff	Instructions
0x80000000	0xffffffff	Stack
0x60000000	0x6ffffff	MPI Accelerator Buffers
0x70000000	0x7ffffff	Dynamic Memory Allocator Buffer

corresponds to the actual MPI buffer of $node_j$. The above are depicted in Figure 4 for $node_{j=0}$ in which the corresponding accelerator buffer addresses are $[0x60000000 : 0x60000000 + N * max_MESSAGE_SIZE - 1]$.

The MPI accelerator is modeled by the OVP Behavioral Modeling (BHM) API and configured by the Peripheral Programming Model (PPM) API. BHM 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 MPI buffer can be identified. For example, assigning ID=3 to an accelerator in a system with 64 nodes resulted in assigning to the local buffer the global address interval of $[0x60000000 + 3 * 64 * MAX_MESSAGE_SIZE : 0x60000000 + 4 * 64 * MAX_MESSAGE_SIZE - 1]$. Although the above can also be assigned by the node's processor during system initialization, in the case of OVP Sim it was required to perform this task during system's instantiation in order to assign certain write function callbacks at the specific addresses. The above callbacks were executed whenever a message was written at the local buffer of the MPI accelerator. This prevented buffer overwrites as each message was instantly consumed by the accelerator which used the dynamic memory allocator (see below) to move the message to a suitable address. The functionality of the callback functions and the MPI accelerator in general were described in C and are analytically described in the following sections. The PPM API allows the peripheral to write to the node's memory by opening an address space to which the peripheral has modify privileges. In addition, the BHM API provides functions for creating ports and nets used for connecting node's devices to each other.

The dynamic memory allocator is used so as to store the asynchronously received MPI messages to a dedicated buffer. In this way the processor does not call any memory allocation functions (i.e., it is not interrupted in any way) while additionally a suitable address for the MPI message is allocated in a few clock cycles. The implemented allocation scheme follows the Buddy allocation algorithm [Knuth 1998]. Buddy was selected as most of its operations are binary and can thus be implemented effectively in hardware.

The bus connects the above nodes to a common address space so that all processor read/write commands either target the memory or directly the MPI accelerator buffers. Specific commands dictate the bus and thus effectively the address space from which the processor should fetch its instructions and store its data. Peripherals connect to the bus through master and slave ports depending on whether they create or respond to bus transactions; for example the processor has a master port initiating bus transactions to read/write to the memory and the memory itself has a slave port as it passively responds to bus transactions reading or writing its contents.

3.3. MPI Library and Benchmarks

A lightweight MPI library was also developed which fully respects the memory mappings presented in Section 3.2. It is based on the openMPI approach [Graham et al. 2006] and implements the functions shown in Table II. These commands suffice for fully executing a significant subset of the NPB suite benchmarks. Additionally, the commands *MPI_Comm.rank* and *MPI_Comm.Size* are implemented which return the

Table II. Implemented MPI Commands

Point-to-Point		Collective
Blocking	Non-Blocking	
MPI_Send	MPI_Irecv	MPI_Alltoall(v)
MPI_Recv		MPI_(All)reduce
MPI_Wait		MPI_Barrier

Fig. 5. Simulation accuracy: average search depth.

ID of the process in the MPI program and the total number of processes instantiated respectively. For the purposes of this work it was assumed that each processor corresponds to a single process and hence to a single ID which was assigned during system's instantiation in OVPsim as described in the last paragraph. The *MPI_Comm_rank* function calls the OVPsim intercepting function *impProcessorId()* which returns the ID of the calling processor.

The IS benchmark of the NPB suite [Bailey et al. 1991] was used as a vehicle to examine the framework's accuracy. IS implements a parallel integer sort algorithm testing the integer computation speed as well as the communication performance of the system under development. The main reason behind our selection was that this application employs more nonblocking MPI commands, in order to achieve asynchronous communication, than all the other ones in the NPB suite.

3.4. Simulation Accuracy

During the development of a SoC, the initial software runs identify the hot spots of the software execution which in turn and based on Amdahl's law, can accelerate the system's overall performance if implemented in a custom-made hardware accelerator. Hence, it is highly desirable for a virtual platform environment like the one used in this work to accurately capture the actual behavior of the system before the actual implementation of the accelerator.

To demonstrate the effectiveness of the proposed framework, a full run of the IS benchmark was performed for both "S" and "W" problem sizes with the system succeeding in sorting the numbers for simulated platforms that utilize from 2 up to 512 nodes. As the focus of this work is on a parallel platform that incorporates certain offloading MPI cores, the characteristics of the MPI's data structures, which are critical for the asynchronous operations (i.e., UMQ and PRQ) were exposed. These results were compared to the ones published in Brightwell and Underwood [2004] and which consist of measurements taken from a real-world parallel system consisting of up to 320 cores.

In this work, PRQ and UMQ maximum lengths and maximum search lengths were shown to grow linearly with the number of processing nodes. This feature was also captured by our runs on top of the proposed framework. A more interesting metric is the average search length of those queues, which in order to be measured accurately, requires the search lengths of all individual list operations. Figure 10 shows the average search depth as measured in [Brightwell and Underwood 2004] and as measured on our experiments on top of the proposed framework. Experimental setups with up to 256 nodes are shown, as the actual real-world system consists of 320 nodes and no further results for larger systems were reported. The results indicate that our high level framework succeeds in capturing the internal behavior of an actual parallel system exposing the search depth as a metric which increases with system size; moreover, our simulation runs on the top of our virtual platform with even more nodes (512) indicate the average search length keeps increasing with system size.

4. BASIC MPI PROCESSOR

Offloading part of the MPI stack to an embedded processor has been proposed as a method to increase the overall performance of a high-end parallel system since such an approach increases the communication-computation overlap. However, there are routines in the MPI stack which, when offloaded to embedded processors, perform so poorly that they actually increase the communication overhead instead of hiding it. One such routine, highlighted in the literature [Brightwell et al. 2005b], is searching the PRQ and the UMQ queues. This section proposes an architecture for an MPI offloading device implementing those MPI tasks that are critical for efficient asynchronous operations. The proposed hardware solution is modeled as a peripheral in the framework described in Section 3, which is also used as a vehicle for further architectural exploration.

4.1. Peripheral Model

The framework presented in Section 3.2 allows integrating under-development hardware within a complete parallel system simulation model. This approach enables the investigation of the behavior of the hardware when it is exposed to complete software applications and not just to specific, corner-case scenarios. As the target platform comprises of a large number of processing nodes, each one executing a specific part of the application as well as actual MPI code, selecting the abstraction layer of the simulations is very crucial; in particular the selection of cycle-accurate simulation would not only require having access to the cycle-accurate model of the hardware which is not available during the initial phases of system's development, but it would also result in extensively long execution times since it requires a cycle-accurate execution of real MPI code concurrently on numerous processing nodes. 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 hidden for the sake of simulation execution time while the hardware model can quite easily be developed.

The basic components of the MPI processor are shown in Figure 6. Their functionality, initially described in C following the untimed functional-level guidelines, is analysed in the following paragraphs.

4.1.1. Message Processor. The message processor orchestrates the data flow through all the MPI processor's components according to the control flow imposed by each MPI command. MPI commands are placed at the message buffers which are connected to either the local bus or to the network. Callbacks associated with message buffers activate the message processor which in turn accesses the buffers so as to decode their fields. The MPI command fields decoded contain the "command type" section which determines the position and the type of the remaining fields in the message buffer.

Fig. 6. MPI Processor model.

Besides the message buffers, the message processor communicates through dedicated ports to the remote MPI buffers and to the local memory. Additionally, the message processor issues certain list operations to the list manager and asks for memory space from the dynamic memory allocator when needed.

4.1.2. Buses and Ports. The MPI processor is connected with the local processor and the local memory through the local bus. The connection between a peripheral and a bus in OVPsim is implemented by a master or a slave port. As each bus is associated with a certain address space, each port is associated with a portion of that space. Hence, a read or write to a port is translated to a read or write to the address interval associated to this port. For example, the local master port is connected to the local bus and is associated with the address interval $[0x70000000 : 0x7fffffff]$, corresponding to the local memory space allocated for storing the MPI message payloads. This connection through a master port allows the MPI processor to initiate both read and write transactions on the local bus which are translated to reading or writing addresses at the local memory.

4.1.3. Message Buffers. Two message buffers are placed at the two slave ports connecting the MPI processor with the local bus and the network. The local slave port corresponds to an address known by the MPI library, so that the execution of an MPI command by the processor is transformed to a simple memory write of the MPI command fields to this address. In this way, the host processor is instantly allowed to continue its computations in the case of a nonblocking command, just after it writes the MPI command to the message buffer.

4.1.4. List Manager. The list manager performs the three basic list operations: search, insert, and delete on the UMQ and the PRQ lists. Each list is implemented by a head and a tail pointer. A free list is maintained including all the MPI buffer positions which are not allocated yet by either of the lists. When a new item is inserted, the free list's head pointer advances to its next pointer's contents and the PRQ or the UMQ tail's next pointer is updated to point to the new element. Accordingly, when an element is removed from the list, the tail of the free list is updated to point to that element. The high-level architecture of our list manager is shown in Figure 7. Unexpected messages are shown in blue, whereas posted receives and free elements are in green and red, respectively. The MPI buffer's portion which is allocated to the PRQ and UMQ lists is partitioned to equally sized segments, each one having fields for a next pointer, message source, and tag as well as for a payload pointer to the memory.

Fig. 7. PRQ and UMQ lists.

4.1.5. Dynamic Memory Allocator. When an unexpected message arrives, its header is either stored in UMQ or thrown away after a match is found in PRQ. The payload carried with the message is not critical during an element search and thus it should not consume valuable space in the MPI buffer. Hence, either the processor should be notified to execute a certain memory allocation routine for the payload, or there should be a mechanism in our MPI processor for allocating and freeing the data corresponding to the payload. As the former would reduce the level of asynchronism in the communication, the latter was selected and a dynamic memory allocation scheme was implemented in the MPI processor. Our allocator is based on the Buddy allocation algorithm [Knuth 1998] which can be very efficiently implemented in hardware since it mainly comprises of binary operations.

4.1.6. MPI Buffer. Searching the PRQ and the UMQ is in the critical path of the send and receive commands, respectively. Hence, accessing the elements of the aforesaid structures should be performed efficiently without having to access the local host memory. As a result, both lists are stored in the MPI buffer which will be implemented on a fast SRAM (much faster than the host DRAM memory). Additionally, allocating and freeing data based on the Buddy allocation scheme requires accessing an arbitrary number of lists, each having an arbitrary number of elements. Thus, if the allocator's data structures are stored in the host DRAM memory, a very high number of bus cycles will be required in order to traverse those allocator's lists. As a result, the allocator's structures are also stored in the MPI buffer, and this significantly increased the allocator's performance as demonstrated in the performance section.

The MPI buffer is modeled as an SRAM inside our novel MPI processor. The size of this SRAM depends on many factors, such as the total number of nodes, the message tag, the maximum number of PRQ and UQM entries, the maximum and minimum size of the dynamically allocated elements, etc. The SRAM size, for an MPI processor within a parallel system containing 16K distinct processing nodes, supporting 16K UMQ and 16K PRQ entries, having 1GB space in local memory for payload allocation while its minimum allocated element is 16Bytes, is shown in Table III.

An example illustrating the functionality of the MPI processor is shown in Figure 8 where an MPI_Irecv command is issued by the processor by writing it at the memory mapped message buffer (step 1). Then, the message processor identifies the type of the command by decoding the message buffer's contents (step 2). As the command is

Table III. MPI Buffer Size for 16K
Unexpected Messages, 16K Posted
Receives and 1GB Allocated Space in Local
Memory

PRQ + UMQ Lists	448 KB
Allocator Lists	96 KB
Total	544 KB

Fig. 8. Nonblocking receive command processing by MPI processor.

a nonblocking receive, the list manager is dictated to search the UMQ for a possible match (step 3). If the list manager finds a match, it copies the message body to the address specified by the value of the “receive address field” and the corresponding entry in UMQ is removed, as shown in Figure 8(b). Otherwise, an entry is inserted in the PRQ list at the fifth step, as shown in Figure 8(a).

5. HASH-BASED MPI PROCESSOR

The PRQ and the UMQ are realized on top of dedicated lists, in all known software and hardware MPI implementations. 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; the latter can be prohibitively high in the case of the recently introduced 1-million nodes systems. One of the most promising solutions to this issue is to utilize a hashing scheme, so that almost constant search times will be enjoyed even at very large systems. The most significant challenges that have not been addressed yet, according to the best of our knowledge, by the already introduced hashing schemes are the following.

- For many applications the average search depth of both PRQ and UMQ is equal to zero which means that the first item is always the one searched for. When the PRQ and UMQ are implemented as linear lists, the search delay for the first item is equal to that needed for accessing the contents of a single pointer. 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.
- In the case where wildcards are not used (i.e., no collective MPI commands are supported), the message source together with a tag can be utilized in a simple hashing

Fig. 9. Hashing scheme for implementing PRQ management.

function. 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 messages, order. However, when wildcards are used, something very common in today's parallel applications, the proposed hashing schemes fail to efficiently identify the matching element, as all the messages of the hash should be searched in order of arrival.

In order to address the issues listed, a novel hashing scheme addressing the requirements of the PRQ and UMQ structures is introduced and demonstrated in Figure 9. Our scheme utilizes the source field of the MPI message as the hashing key. The hashing function assumes that the number of available buckets is a power of two, leading to an efficient hardware implementation. For a hashing scheme with M buckets, a message with source field 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 $i \& (M - 1)$. The above function is analyzed to a binary "&" and a subtraction by 1 (assuming a RISC embedded processor). However, in hardware no logic is needed, as the hashing function reduces to selecting the M LSBs of i . The above indicates that the hardware implementation of the proposed hashing scheme results in negligible overhead when compared to conventional linear list hardware implementations.

With respect to wildcards, a separate list is maintained which stores the entries in the order they arrived. In case no elements with wildcarded source nodes have arrived, the ordinary hashing scheme of the last paragraph is applied; in case there are wildcarded elements this separate order list, is utilized. In this latter case, if an element is found and removed from the order list the chain value in the hash in which the element was also assigned should also be updated. However, the element visited before the match in the order list can be different from the previous element in the specific bucket's chain. In order to further demonstrate this problem consider the example shown in Figure 9 which has wildcarded elements, the order list is traversed and the element with source=33 matches the search and should be removed. The previous element in

Fig. 10. Maximum and average search depth with respect to number of buckets for 64 procs.

the order list is the one with source=48 which is updated so that its next field points to the element with source=38. If there is no previous field in the hash lists, then there would be no way to perform such an update.

Storing the next, previous, and payload pointers in the MPI buffer would be a waste of space as it is desirable to keep in the fast memory only the elements which are used by the list search operation. Hence, the next pointers of the lists as well as the message source and tag fields are stored in the MPI buffer whereas the fields which are updated only upon a message matching are stored in the local host memory. It should be noted that there is no need for the MPI processor to export the value of a pointer to the local memory as fields of the same message are stored at the same index of the two arrays. Hence, if the source field of an MPI message is stored in the 3rd entry of the MPI buffer then the data pointer field of the same MPI message should be at the 3rd entry of the host memory array which stores the MPI messages.

In order to evaluate the effectiveness of our hashing scheme, the complete MPI processor model was integrated in the platform and the IS benchmark was utilized in our experiments. In our experiments, 64 nodes were instantiated and hashing schemes with 1, 2, 4, 8, and 16 buckets were simulated. The configuration with a single bucket matches the one of the conventional serialized list-based scheme. Figure 10 shows the results of the maximum search depth and the average search depth of the PRQ when the complete benchmark is executed. The serialized list-based solution has worst-case depth of 64 entries. Increasing the number of buckets by a factor of 4 results in 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. As this same figure clearly demonstrates, for the conventional list-based approach, 15 entries were accessed in average, a number which is reduced to 6.4, 2.9, 1.3, and 0.6 for configurations with 2, 4, 8, and 16 buckets respectively. As a result we claim that our hashing scheme can accelerate the searching of the queues by a factor from 2 to about 30 depending on the number of buckets.

6. HARDWARE IMPLEMENTATION

The cycle-accurate model of our MPI processor was synthesized using the Cadence [2012] C-to-Silicon high-level synthesis tool, version 10.10. utilizing the concepts described in the Appendix. The automatically generated HDL was synthesized with Cadence RTL Compiler version 10.1 and simulations were performed with Cadence ncsim

Table IV. Post-Synthesis Performance, Area and Power Results of MPI Processor Components Using a 40nm Standard Cell Technology Library

Module	Frequency(GHz)	Area(μm^2)	Power(mW)
PRQ + UMQ List Manager	1	95055	39.08
Dynamic Memory Allocator	0.55	101254	24.95

version 8.20. The silicon results obtained when utilizing a modern 40nm standard cell technology library are shown in Table IV. Based on those figures it is clear that our module can be instantiated at even a low-cost embedded parallel framework, while its complexity is orders of magnitude smaller than even a very simple CPU. Furthermore, the power consumption of the proposed device is at least an order of magnitude lower than that of even a low-power CPU (i.e., the recently introduced ARM-based Cortex A9 implemented on a 32nm CMOS technology) making it ideal for the recently introduced embedded parallel systems utilizing the MPI protocol [Go andhringer et al. 2010].

7. PERFORMANCE

In a significant number of real-world parallel applications the size of the MPI's queues grows linearly with the number of nodes in the parallel system. Our measurements have confirmed that IS, one of the NPB suite benchmarks, follows the above trend for both PRQ and UMQ and in particular the number of nodes in the system is equal to the maximum number of messages residing in the PRQ and UMQ structures. Looking at the reasoning behind this feature, we realized that when a node waits for messages from a certain subset of the other nodes it is not possible to know a priori their arrival order. The above uncertainty has been shown to increase with system's size while the higher the number of nodes the more difficult to load-balance the application on them. Hence, 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. Then we compare the performance of our novel system with that of a high-end Intel CPU [Intel 2008] as well as with an ARM A8 [ARM 2012]¹ state-of-the-art embedded processor executing the exact same MPI tasks (from the most widely used openMPI library) on top of the Fedora 14 Linux OS. Finally we demonstrate the performance gain of our approach when compared with the only similar hardware system presented so far.

Underwood and Brightwell [2004] introduced the so-called “preposted latency benchmark” in order to be able to analytically examine the latency triggered when unloading the PRQ; this latency is probably one of the most important performance metrics for any MPI implementation. This benchmark is an enhanced version of the ping-pong one which is widely used as the basis for the latency measurements in typical systems supporting the MPI intercommunication protocol. The preposted benchmark builds a PRQ of a specified number of entries and then posts an unexpected message which matches an element residing at a predefined queue location (given as a percentage of the total queue length). In this way, the MPI protocol is forced to traverse a predefined percentage of the list and not only its first entry; once a match is found at the first entry the latency overhead is obviously the lowest possible.

In this work we extend the preposted latency benchmark so that the PRQ is also built with a predefined number of entries. Moreover, in our case the application is forced to

¹The ARM A8 was simulated with a clock of 1.5 GHz, matching the frequency of the TI Sitara [TI 2012] implementation of the A8, the fastest in the market to the best of our knowledge.

(a) performance of Intel E8400, ARM A8 and our basic MPI processor

(b) performance of ARM A8 with hash-based MPI software and of our hash-based MPI processor

Fig. 11. Intel E8400, ARM A8, and MPI processor performance comparison with respect to the “preposted latency benchmark”. x and y axis are in logarithmic scale.

always match the last element in the list and hence the total number of items traversed is $n + (n - 1) + \dots + 2 + 1 = (n * (n + 1))/2$.

In Figure 11(a) our Basic MPI processor (i.e., no hashing is utilized) is compared with both a high-end host processor as well as a state-of-the-art embedded one executing the abovementioned benchmark when no hashing is introduced. The high-end processor performance is measured using the Intel VTune [VTune 2012] profiler. The embedded processor is evaluated using the API provided by the OVP simulation tool. More specifically, a word allocated in the memory is written at the beginning and at the end of each queue processing routine. Memory callbacks, provided by the OVP simulation environment, are triggered at each access to the above memory location, measuring exactly the number of cycles consumed.

Fig. 12. Intel E8400, ARM A8 and MPI processor performance comparison on unloading an MPI queue of more than 32K entries. y axis is in logarithmic scale.

Results show that for small queue sizes our novel MPI processor is 1 to 2 orders of magnitude faster than both the high-end processor and the embedded CPU.²

Moreover in Figure 11(b) our MPI hash-based processor is compared with the ARM A8 embedded state-of-the-art CPU executing the same benchmark while also utilizing the proposed hashing scheme in software; as those results clearly demonstrate our novel device is 2 orders of magnitude faster than the specified CPU.

In both figures, the small leftmost embedded figure highlights the unload times of the queues with 256 up to 2048 entries. Since for numerous applications, as described in Brightwell et al. [2006], traversing a small number of elements is the common case, so the results presented at those zoom-outs are very important; even for those small queues our systems are from 1 to 2 orders of magnitude faster than the corresponding software solutions.

The other embedded figures are zoom-outs which show the unload times for queues which fit in the MPI buffer (i.e., up to 16K) and which in turn is mapped to a 512KB SRAM with 1ns clock which needs 2 cycles per access (such an SRAM was trivially synthesized as described in Sym [2012]). Those results clearly demonstrate that our pioneering MPI processor enjoys at least one order of magnitude less latency than that of the high-end Intel processor when unloading the specified queues.

Results spanning from 32K entries and up to 128K entries highlight the very long time required by any CPU in order to unload the queue; based on the fact that, as it was described before, for certain applications the number of the queue entries is equal to the number of nodes in the parallel system, this measurement clearly demonstrates the unfeasibility of the pure software approach for parallel systems comprising of hundreds of thousands of nodes.

Figure 11 assumes that for large queues an 8MB memory module with a latency of 2ns has also been synthesized and integrated to the MPI-processor module chip. That obviously increases the silicon cost of the complete device considerably (by about an order of magnitude). In order to implement a significantly lower-cost device our novel

²The high-end processor suffers from performance degradation after 16384 messages are stored due to high number of cache misses. On the other side, for the simulated embedded processor, an extremely big cache is assumed for the sake of simulation speed and of our aim to get the best possible results for the software implementation.

Fig. 13. Unload time for systems with N nodes where $N = \{ 512, 8K, 64K \}$, when using hashing scheme with M buckets where $M = \{ 1, N/4, N/8 \}$. Both x-axis and y-axis are in logarithmic scale.

Fig. 14. Queue processing speedup for 4 NPBs.

MPI processor (without incorporating any on-chip memory) can be connected to an off-chip 64Mbit SRAM [Cyp 2012] which has a latency of 12 ns or in other words six times the access time of the corresponding on-chip SRAM. Figure 12 focuses on queue sizes requiring off-chip memory for both the basic and the hash-based MPI processor and compares their performance with the high-end as well as the embedded processors. Based on those measurements it is clear that our MPI processor outperforms the software solutions for the large queues by two orders of magnitude even when an external SRAM has been utilized.

In order to further investigate the performance of our hashing schemes we have conducted several experiments with different number of buckets and different number of nodes. As the graphs in Figure 13 clearly demonstrate the latency triggered by our MPI processor is decreased by an order of magnitude if 128 buckets are utilized in a 512-node system, two orders of magnitude if 2K buckets are employed in an 8K-node system, and three orders of magnitude when we have 8K buckets in a 64K-node framework, when compared with the single-bucket system.

Furthermore, we evaluated the performance of our novel approach when executing 4 different benchmarks of the NAS benchmark suite. The first graph in Figure 14(a) shows the speedup triggered on the MPI processing, when compared with the ARM A8 embedded CPU, by both the basic hardware device as well as the hash-based module

Fig. 15. Overall processing speedup (%) for 256 processors (several computation kernels have been deactivated).

when the four benchmarks are executed on a 32-CPU parallel system. As this graph demonstrates the speedup of our high-end MPI accelerator when compared with the software hash-based approach varies from 16x to over 20x. When moving to a larger system this speedup grows even more as demonstrated in Figure 14(b); in particular it varies from 41x to 119x.

It should be noted that those speedup numbers cover the actual MPI processing tasks in both the ARM8-based system and our hardware co-processors. Those numbers are important since, as described in Brightwell and Underwood [2004], each application poses different demands with regards to the scanning of the MPI queues. So even though two applications may have the same average number of elements in their MPI queues (let's say N), one may have to traverse, in average, just one or two of them in order to find the requested data item whereas another application may have to traverse, in average, up to N items. This is the reason that, for the different NAS benchmarks, we get different speedups.

Moving to the actual overall processing time of the applications, Figure 15 demonstrates the speedup triggered by: (a) the software-based approach utilizing our hash function, (b) our hardware accelerator without hashing, and (c) our hardware accelerator utilizing our hashing scheme. Since those numbers have been derived by instruction-accurate simulations of our hardware systems within a 256-node platform executing complete applications, certain processing kernels, which do not affect the communication patterns, have been disabled so that the simulation could complete in reasonable time. As those numbers clearly demonstrate, the overall speedup of the applications, due to our novel hardware scheme, can be up to up to 20%. Obviously those numbers differ significantly from one application to another since they are affected by the percentage of the time each program spends on the intercommunication tasks (i.e., the more the intercommunication, when compared with the actual processing, the better our results). Since the intercommunication percentage increases with the number of nodes [Brightwell and Underwood 2004] the overall speedup is expected to be much higher in multi-thousand node systems.

Moving to similar hardware systems, Underwood et al. [2005b] is the only one describing an architecture for a module which can accelerate the list handling of MPI. In particular, the paper proposes to enrich the embedded processors running the MPI stack with an accelerator, namely list manager, able to perform the basic list operations on the UMQ and PRQ. It was assumed that the message headers could be stored in an SRAM module so that the main memory was accessed only upon an entry match. The proposed solution was compared to a software solution executed on an embedded processor and it was estimated, using simulations, that the latency due to MPI queue searches can be reduced by a factor of 80% if tens of entries are traversed each time an eager send or receive is issued. The list manager was not synthesized to real hardware

Fig. 16. Underwood's implementation and hash-based MPI processor performance comparison on unloading an MPI queue.

and it was not integrated to any parallel platform. Also the dynamic memory allocation task has not been investigated (which in our case proved to complicate and delay the actual processing) while all the presented results stemmed from small synthetic benchmarks. However, our simple (i.e., without hashing) module can be considered as a sophisticated implementation of the presented in Underwood et al. [2005b] high-level architecture, so Figure 14 as well as Figure 16 also demonstrate the speedup of our novel device when compared with an advanced implementation of Underwood's architecture. Based on the presented results for the four NAS benchmarks our module is from 30% to over 250% faster than Underwood's approach for small parallel systems while our speedup is from 10x to 20x in the case of relatively large parallel frameworks. Moreover, when looking at the micro-benchmark numbers our system is from 20% and up to two orders of magnitude faster than Underwood's solution.

As all our performance results demonstrate our novel system clearly outperforms any existing approaches while it scales very well with the number of nodes, the speedup against the existing solution increases with the number of queue entries and thus, effectively, with the number of nodes in the parallel system. This feature makes our device an ideal candidate for the recently introduced million-nodes parallel systems.

8. RELATED WORK

The major objective for all related approaches which efficiently implement the MPI asynchronous operations is to offer an effective infrastructure supporting real and extended computation and communication overlap. To achieve that, the main processor should not be involved in any data exchange focusing entirely on its data processing. Hence, the MPI communication stack is offloaded to separate main processors, co-processors, embedded processors on the NIC, or even to dedicated hardware. This section reviews the above solutions introduced so far in the literature and highlights their differences with regards to the one proposed in this article.

As described in the last section Underwood's approach (Underwood et al. [2005b]) can be considered similar to ours; our performance results clearly demonstrated that we outperform it, especially for medium and large parallel systems.

In Almási et al. [2004] different schemes for accelerating the MPI communication on the BlueGene/L highly parallel system were introduced. It was reported that the highest communication bandwidth is achieved when one of the two processors on each computation node is running the MPI stack whereas the other performs the actual data processing. Scaling the above finding to the complete system requires half of the

processing power to be assigned only to the execution of the MPI stack. Obviously, this is a software approach which still has the disadvantage of increased latency, in comparison with our dedicated hardware, as the last section clearly demonstrates, whereas our hardware accelerator has about 1/100th of the silicon cost of the CPU executing the MPI stack.

Another option for accelerating the MPI execution is to offload the MPI processing to the NIC processor. Although the processing power of such processors is limited when compared with that of the host processors, their proximity to the network as well as to the resources of the NIC makes them good candidates for the MPI offloading task. An example is the Lanai Z8ES chip embedded in Myricom's [2009] NICs. Myricom provides software which enables offloading the whole MPI stack onto Lanai's embedded processor. Quadrics has introduced another MPI implementation in the Elan4 [Petrini et al. 2002] high-performance NIC. Elan4 is, to the best of our knowledge, the only offloading solution using a hashing scheme [Underwood and Brightwell 2004]. As described in Section 4, the implementation of hashing in the NIC's embedded processor triggers a constant latency when searching deep in PRQ and UMQ (if wildcards are used the search time grows as with the list implementations), but severely affects the search time when the first element is picked from the list this is shown to be the common case in MPI programs [Brightwell et al. 2006]. Again those approaches are software based ones and suffer from the same drawbacks, when compared with our dedicated hardware solution, as in the BlueGene case described in the last paragraph.

The initial Underwood's list manager was later augmented by an associative matching unit in Underwood et al. [2005a]. As the MPI queue processing has to be performed in order, and wildcards should also be supported, state-of-the-art FPGA-based TCAMs have been proposed as an efficient alternative to the hashing schemes. Hence, a novel architecture was presented which can search the whole queue in a constant number of few clock cycles. The most significant drawbacks of the proposed solution were the high silicon cost, the lack of scalability as well as the power consumption. Two associative units of only 256 entries each, could marginally fit on a relatively big, at that time, FPGA. In addition, searching the associative unit triggers significant power consumption, as a big portion of the entries are concurrently accessed. More importantly, this approach cannot scale to today's and tomorrow's multi-thousand and million-nodes systems since; based on an extrapolation from the numbers presented in the paper, it requires a complete current state-of-the-art FPGA to be attached to each processing node for every 4K processing nodes of the parallel system; as a result even in a simple 16K-nodes system each node should be attached to 4 high-end-FPGAs resulting in a total of 64K large FPGAs implementing the queue searches.

In Hemmert et al. [2007] a programmable architecture is presented which executes MPI's lower level matching code as specified in MPI's Abstract Device Interface (ADI). The architecture is comprised of two parallel pipelines executing bitwise, logical, arithmetic and permutation commands. It was shown that the above microcoded architecture outperforms software based solutions. When compared to a list manager architecture, as the one described in Underwood et al. [2005b], their approach triggers performance and area penalties of 6%–10% and 30% respectively; the programmability, so as to support different queue organizations is their major advantage. However, nothing has changed, with respect to queue entries matching, since the first version of the MPI standard and no changes have even been proposed to this task, therefore the programmability offered has not proved to be important so far.

The work in Tanabe et al. [2009] introduced a hardware accelerator for eager MPI messages exhibiting a much lower latency than the conventional software approach. The headers of eagerly sent messages are buffered in an SRAM and the message

bodies are stored at a dedicated DDR memory. Message bodies are grouped in separate buffer spaces in the DDR memory, according to their source ID in order to achieve better spatial locality. In addition, packing and unpacking commands are introduced, which reduce data striding and effectively the time to load and store the corresponding messages. Although the above architecture does not directly accelerate the message matching itself, a speedup with respect to a software solution is expected due to its sophisticated caching scheme which significantly decreases the accesses to the main memory. This work does not present any performance or hardware implementation results whereas it is expected that the presented system will be not be very efficient in large parallel systems unless it is combined with a hashing system like the one we propose.

Finally, certain High-Performance Computing (HPC) systems [Fahey et al. 2004] incorporate reconfigurable hardware which can be utilized so as to accelerate either data or communication intensive applications. In Saldana and Chow [2006] an architecture implementing the MPI stack on an FPGA was introduced. The queue matching was performed by traversing linearly the UMQ and PRQ. Therefore, the main difference with our approach is that no hashing has been utilized while the memory allocation scheme is not presented. Moreover, no actual performance results for the traversing of the queues nor of the silicon resources needed are presented, so as to be able to compare it with our hardware approach.

9. CONCLUSIONS

One of the main problems in recently introduced million-nodes parallel systems is the high intercommunication delay. This intercommunication delay can be hidden if asynchronous communication primitives are utilized. However, even in this case the processor node should keep track of the status of the various messages sent to and/or received from those million nodes while also processing the actual sends and receives. This is a very time-consuming task therefore there are several approaches trying to offload it from the main processor.

This article presents how the intercommunication tasks, that are handling the sends and receives of the intercommunication messages to/from the other nodes, can be efficiently implemented in custom hardware. Although a similar, yet simpler approach has been proposed in the past, this is the first known system that: (a) employs, in custom hardware, a novel scheme for reducing the search time in the various send and receive queues by up to three orders of magnitude, (b) has been actually synthesized (and not only modeled at the architectural level) in a state-of-the-art CMOS technology, (c) whose actual hardware performance and power consumption is demonstrated and (d) has been profiled on real-world applications (and not small synthetic benchmarks) running on top of a pioneering parallel-systems simulator.

Our accelerator is from one and up to four orders of magnitude faster than two general-purpose CPUs executing the same tasks, it is from 20% to two orders of magnitude faster than the only similar hardware approach while its speedup grows with the number of nodes in the parallel system. Based on those results we believe that our approach might probably be the only one that can handle the MPI latency, in the recently introduced million-node systems, within realistic timeframes. Moreover, our device consumes 100 times less power and it is being implemented at 1/100th of the silicon area of a small embedded CPU. Based on those figures, we believe that the presented system can be utilized not only on HPC frameworks but also on the future highly parallel embedded systems, which up to now were not using the popular MPI protocol due to its high overhead when executed on an embedded CPU.

ACKNOWLEDGMENTS

The authors would like to thank Dimitrios S. Nikolopoulos for his very constructive suggestions and Nikos Taburatzis for porting two of the NPB benchmarks to our embedded platform.

REFERENCES

- ALMÁSI, G., ARCHER, C., CASTAÑOS, J. G., ERWAY, C. C., HEIDELBERGER, P., MARTORELL, X., MOREIRA, J. E., PINNOW, K. W., RATTERMAN, J., SMEDS, N., STEINMACHER-BUROW, B. D., GROPP, W., AND TOONEN, B. R. 2004. Implementing MPI on the Bluegene/L supercomputer. In *Proceeding of the 10th International Euro-pav Conference*. 833–845.
- ARM. 2012. <http://www.arm.com/products/processors/cortex-a/cortex-a8.php>.
- BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATKRISHNAN, V., AND WEERATUNGA, S. K. 1991. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. ACM, New York, 158–165.
- BORKAR, S. AND CHIEN, A. A. 2011. The future of microprocessors. *Comm. ACM* 54, 5, 67–77.
- BRIGHTWELL, R., BARRETT, B., HEMMERT, K., AND UNDERWOOD, K. 2010. Challenges for high-performance networking for exascale computing. In *Proceeding of the 19th International Conference on Computer Communications and Networks (ICCCN)*, 2010. 1–6.
- BRIGHTWELL, R., GOUDY, S., AND UNDERWOOD, K. D. 2005a. A preliminary analysis of the MPI queue characteristics of several applications. In *Proceeding of the International Conference on Parallel Processing*. 175–183.
- BRIGHTWELL, R., GOUDY, S. P., RODRIGUES, A., AND UNDERWOOD, K. D. 2006. Implications of application usage characteristics for collective communication offload. *Int. J. High Perform. Comput. Netw.* 4, 3/4, 104–116.
- BRIGHTWELL, R., RIESEN, R., AND UNDERWOOD, K. D. 2005b. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.* 19, 2, 103–117.
- BRIGHTWELL, R. AND UNDERWOOD, K. D. 2004. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the International Parallel and Distributed Processing Symposium* 9.
- CADENCE. 2012. <http://www.cadence.com>.
- CYPRESS. 2012. <http://www.cypress.com/?mpn=CY7C1089DV33-12BAXI>.
- CAI, L. AND GAJSKI, D. 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.
- FAHEY, M., ALAM, S., DUNIGAN, T., VETTER, J., AND WORLEY, P. 2004. Early evaluation of the Better Cray XD1. Tech. rep., Oak Ridge National Laboratory.
- FORUM, M. P. 1994. Mpi: A message-passing interface standard. Tech. rep., Knoxville, TN, USA.
- GO ANDHRINGER, D., HU ANDBNER, M., HUGOT-DERVILLE, L., AND BECKER, J. 2010. Message passing interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, 357–364.
- GRAHAM, R. L., WOODALL, T. S., AND SQUYRES, J. M. 2006. Open MPI: a flexible high performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM'05)*. 228–239.
- HEMMERT, K., UNDERWOOD, K., AND RODRIGUES, A. 2007. An architecture to perform NIC based MPI matching. In *Proceedings of the IEEE International Conference Cluster Computing*. 211–221.
- IMPERAS. 2012. <http://www.imperas.com>.
- INTEL. 2008. <http://www.intel.com>.
- INTEL. 2012. <http://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- KELLER, R. AND GRAHAM, R. L. 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.
- KNUTH, D. E. 1998. *The Art of Computer Programming, vol. 1 (3rd ed.): Fundamental Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.
- MYRICOM. 2009. <http://www.myricom.com>.
- OPEN RISC. 2012. <http://opencores.org/openrisc,architecture>.
- OVP. 2012. <http://www.ovpworld.org>.

- PETRINI, F., FENG, W.-C., HOISIE, A., COLL, S., AND FRACHTENBERG, E. 2002. The Quadrics network: high-performance clustering technology. *Proceedings of the 15th International Conference on Micro*, 22, 1, 46–57.
- SALDANA, M. AND CHOW, P. 2006. TMD-MPI: An MPI Implementation for multiple processors across multiple FPGAs. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL'06)*. 1–6.
- SYNOPSYS. 2012. Synopsys high density leakage control SRAM 512K sync compiler, GlobalFoundries 40LP P-optional Vt/cell SVt S-bitCell. <http://www.synopsys.com>.
- TANABE, N., OHTA, A., WASKITO, P., AND NAKAJO, H. 2009. Network interface architecture for scalable message queue processing. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*. 268–275.
- TI. 2012. www.ti.com.
- UNDERWOOD, K. AND BRIGHTWELL, R. 2004. The impact of MPI queue usage on message latency. In *Proceedings of the International Conference on Parallel Processing*. 152–160.
- UNDERWOOD, K. D., HEMMERT, K. S., RODRIGUES, A., MURPHY, R. C., AND BRIGHTWELL, R. 2005a. A hardware acceleration unit for MPI queue processing. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*.
- UNDERWOOD, K. D., RODRIGUES, A., AND HEMMEIT, K. S. 2005b. Accelerating list management for MPI. In *Proceedings of the IEEE Symposium on Cluster Computing*. 1–10.

Received June 2012; revised November 2012; accepted November 2012