Abstract—This paper presents several challenges and solutions in designing an efficient Message Passing Interface (MPI) implementation for embedded FPGA applications. Popular MPI implementations are designed for general-purpose computers which have significantly different properties and trade-offs than embedded platforms. Our work focuses on two types of interactions that are not present in typical MPI implementations. First, a number of improvements designed to accelerate software-hardware interactions are introduced, including a Direct Memory Access (DMA) engine with MPI functionality; the use of non-interrupting, non-blocking messages; and a proposed function, called MPI_Coalesce, to reduce the function call overhead from a series of sequential messages. These improvements resulted in a speed-up of 5-fold compared to an embedded software-only MPI implementation. Next, a novel dataflow message passing model is presented for hardware-hardware interactions to overcome the limitations of atomic messages, allowing hardware engines to communicate and compute simultaneously. This dataflow model provides a natural method for hardware designers to build high-performance, MPI systems. Finally, two hardware cores, Tee cores and message watchdog timers, are introduced to provide a transparent method of debugging hardware MPI designs.

I. INTRODUCTION

The Message Passing Interface (MPI) is a popular programming model for distributed memory systems and is often used in High Performance Computing (HPC), such as cluster-based systems. As a result, MPI Application Programming Interfaces (APIs) are generally implemented to suit the properties of the HPC cluster paradigm, which uses relatively high-latency communication networks and relies on significant amounts of memory for each node.

Although MPI is generally used for HPC clusters, there is growing interest in using MPI for embedded applications [1], [2], [3], [4], [5], [6]. Current Field Programmable Gate Arrays (FPGAs) are now sufficiently large that entire high-performance systems of embedded processors, memory units and computational cores can be integrated in a single chip. MPI provides an efficient and portable programming model that uses a common interface to unify these modular, heterogeneous systems. Various embedded MPI implementations have been designed for processors in FPGAs. The model is further complemented by hardware cores that implement MPI functionality, allowing for seamless integration of hardware engines [1]. This abstraction of MPI processes, called ranks, allows for rapid prototyping of entire systems where processors can be replaced with hardware cores for increased performance without modifying the message passing model.

However, the two systems, HPC clusters and embedded FPGAs, have drastically different properties and efficient MPI implementations must account for the various bottlenecks and trade-offs. HPC clusters are composed of powerful processors, each with access to plenty of memory organized in a complex hierarchy of caches. For the communication, HPC clusters use a variety of interconnects that require robust protocols, resulting in a relatively high-latency communication network. In contrast, embedded FPGAs use simpler processors running at a slower clock rate with a limited amount of memory. Communication is completed via networks of on-chip, low-latency, high-bandwidth links. In addition, embedded FPGA communication protocols use significantly less overhead than HPC implementations since the on-chip links are more reliable and do not require redundant protocols, such as error checking codes. Although there are efficient MPI implementations for HPC clusters, a direct port of these implementations are ineffective for embedded FPGAs as a result of their significantly different properties, summarized in Table I.

<table>
<thead>
<tr>
<th>Property</th>
<th>HPC Cluster</th>
<th>Embedded FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2-3 GHz</td>
<td>100-200 MHz</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>&gt;1 GB</td>
<td>1-20 MB</td>
</tr>
<tr>
<td>Memory Size</td>
<td>10 GPUs (20k cycles)</td>
<td>None</td>
</tr>
<tr>
<td>Interconnect</td>
<td>125 MB/s</td>
<td>400-800 MB/s</td>
</tr>
<tr>
<td>Latency</td>
<td>10ns (10 cycles)</td>
<td>100ns (10 cycles)</td>
</tr>
<tr>
<td>Bandwidth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Furthermore, embedded FPGA MPI implementations require extra consideration due to the range of heterogeneous environments. Embedded FPGAs consist of ranks implemented as both software processes and hardware engines; each implementation introduces a new set of challenges. Three classes of interactions must be considered:

- **Class I: Software-software interactions** – Interactions between groups of embedded processors. The challenge
in this class is the design of an embedded processor MPI
implementation with small memory footprint, which has
been thoroughly investigated [1], [2], and thus, is not
further discussed in this paper.
- **Class II: Software-hardware interactions** – Interactions
between embedded processors and hardware engines. The
challenges in this class arise from the heterogeneity of the
system: the hardware engines can handle messages faster
than processors. As a result, special considerations and
optimizations must be made to increase the performance
of the processor-related MPI functionality.
- **Class III: Hardware-hardware interactions** – Interac-
tions between groups of hardware engines. The challenge
is to maximize the performance of hardware engines,
which unlike their processor counterparts, are capable of
handling communication and computation concurrently.

This paper explores the challenges and solutions in designing
an MPI implementation that is amenable for embedded ap-
lications that use both processors as well as hardware engines.
Optimizations for embedded processor MPI implementations
will be presented and a new dataflow message passing model
for hardware engines will be explored. In addition, a set of
debugging interfaces that can be used to analyze embedded
MPI implementations will be discussed.

The rest of the paper is organized as follows: Section II
contrasts existing work with our research. Sections III and IV
investigate the challenges of Class II and III interactions,
respectively. Section V describes custom debugging hardware
cores used to analyze embedded MPI designs. Section VI
quantifies the performance effects of the optimizations and
the paper is concluded in Section VII.

II. RELATED WORK

MPI has been made available to programmers by computer
manufacturers who provide an optimized library for their own
architectures for example Intel-MPI [7] or SGI’s MPI [8].
Alternatively, there are well-known open source MPI imple-
mentations, such as MPICH [9] or OpenMPI [10]. The success
of MPI as a scalable, portable and relatively easy to learn
programming model has caused researchers to propose MPI
for high-performance embedded systems.

Recent research [1], [2], [3], [4], [5], [6] describe proof-of-
concept MPI implementations targeting embedded systems,
showing an increasing interest in the topic. These implementa-
tions have a varying degree of functionality and requirements.
These papers also discuss different ways to address the limita-
tions found in typical embedded systems. For example, in the
eMPI/empich project [2], the main focus is to port MPICH
to an embedded platform and reduce its memory footprint by
removing some MPI functions. Azequia-MPI [3] is an MPI
implementation that uses threads instead of processes making
MPI applications more lightweight, however, it requires an
operating system that supports threads, which in embedded
systems it is not always available. The SoC-MPI [4] library
is part of a project that focuses on automatic generation of
Multiprocessor-System-on-Chip architectures that uses MPI as
a programming model. In this work, the authors take advantage of the flexibility of FPGAs to provide hardware support for
broadcast operations. Indeed, the flexibility to experiment with
hardware support is a key advantage of FPGA-based embed-
ded systems compared to HPC systems. In fact, by exposing
MPI to the embedded world, new requirements may arise, such
as fine-granularity data-flow communication patterns, typical
in hardware design. In this paper, we also use hardware support
to improve MPI performance to address such requirements
using TMD-MPI [1].

TMD-MPI is a subset implementation of the MPI standard
that allows hardware engines implemented in FPGAs to be part
of the message-passing network simplifying the programming
model and integration of such engines into a parallel appli-
cation. TMD-MPI has been ported to the following boards:
Amirix PCI [11], BEE2 [12] and recently to the Xilinx
ACP (available through Nallatech[13]), and it can run on the
following processors: MicroBlaze, PowerPC and Intel X86.
TMD-MPI is beyond the proof-of-concept stage and is now
at the optimization stage. In this paper, our optimizations are
built upon the framework of TMD-MPI.

III. CLASS II: PROCESSOR-BASED OPTIMIZATIONS

A. Background

This section discusses the optimizations used to increase
the performance of the MPI implementations on embedded
processors, particularly when interacting with hardware en-
gines (Class II interactions). Before describing the Class II
optimizations, it is important to understand why the HPC
message paradigm is not suitable for embedded applications.
Class I interactions are commonly designed for the HPC
cluster paradigm, which utilizes operating systems, powerful
processors and significant amounts of memory. However, the
communication latency for each message is relatively high. As
a result, the typical message passing paradigm prefers to com-
bine data into fewer large messages, which are subsequently
stored in memory and retrieved at a much later time.

However, embedded FPGA designs present a different trade-
off with respect to memory and communication latency. Em-
bded systems have limited memory resources and as a result,
the message passing paradigm of storing large messages for
future use is not appropriate. Instead, a 'just-in-time’ paradigm
is more suitable for both embedded Class I and II interactions:
frequently sending short messages with just enough data
to complete a single unit of computation. This maintains
an effective communication-to-computation ratio due to low-
latency links while providing better resource utilization.

In addition to the resource motivation, the interaction class
must consider the widely varying use case that arises from
each platform. Class I software-software interactions are fairly
homogeneous, which contrasts against the heterogeneous na-
ture of Class II software-hardware interactions. The hardware
engines are optimized for computationally intensive roles and
are able to communicate and compute much faster than their
software counterparts. However, embedded processors still
play a vital role in FPGA designs as control and memory
distribution units, and perform very little computation.

The ‘just-in-time’ paradigm and Class II interactions use
case gives rise to a producer-consumer model where hardware
engines wait to consume messages produced by the processor.
Since the hardware engines are responsible for the majority of the computation and consume faster than processors produce, the idle phase of a hardware engine should be minimized. Thus, special considerations and optimizations are required for embedded processors in these Class II interactions.

B. Direct Memory Access MPI Hardware Engine

The first optimization for Class II interactions is the design of a hardware engine that provides MPI functionality through Direct Memory Access (DMA). Rather than implementing MPI solely in the software framework, a significant portion of the traditional progress engine functionality is offloaded to a hardware core. By designing DMA hardware that can complete MPI messages, time consuming tasks, such as memory transfers, are completed faster in hardware, freeing the processor to execute other commands.

Our DMA hardware implementation is designed for Xilinx FPGAs and is called the PLB Message Passing Engine (PLB_MPE). It interfaces with the Processor Local Bus (PLB), where it is both a slave and master, and a set of Fast Simplex Links (FSLs), a unidirectional point-to-point communication bus. The PLB slave interface consists of two memory-mapped hardware queues for pending transmit and receive messages, respectively. Upon calling a MPI function, the processor writes the message information as four words – which consists of the source or destination rank, the memory location of the message buffer, the message size, as well as the message tag – to the appropriate queue. The PLB_MPE then performs the memory copy between the PLB memory and the on-chip network. This implementation is shown in Fig. 1.

The PLB_MPE hardware core provides two important advantages for Class II interactions. First, the DMA-capable PLB master is able to complete memory transactions faster than an embedded processor since it can execute burst memory transactions and does not require the overhead of instruction decoding. Thus, the effective bandwidth for data transmission is significantly increased.

Next, the use of hardware queues to buffer message requests provides significant performance advantages. The processor only needs to write four words to the PLB_MPE for each MPI message, regardless of the message size. The constant time required per DMA message as well as the ability to queue messages provides an important functional advantage for the PLB_MPE. Hardware compute engines rotate between the two phases: computation (busy) and communication (idle).

Because the processors are slower than hardware engines, the ability of the PLB_MPE to decouple the message request from the message transfer allows processors to queue message requests while the hardware engines are busy so that message transfers occur immediately once the hardware engines are available. This minimizes hardware idle time and maximizes performance.

It should be noted that the DMA functionality is completely transparent to the user: the user simply calls MPI functions and the DMA setup is handled by the implementation.

C. Non-Interrupting, Non-Blocking MPI Functions

The next consideration for Class II interactions is the use of non-interrupting, non-blocking MPI functions. MPI messages are divided into blocking and non-blocking counterparts. Blocking function calls only return once the message buffer can be safely reused, while non-blocking function calls return immediately, allowing the processor to continue regardless of the message status. For non-blocking functions, a request handle must be provided so that the status of the pending message can be determined at a later stage. Class I software-software interactions generally use non-blocking functions and request handles to effectively overlap communication and computation (Fig. 2 – example 1).

```c
/*Example 1: Interrupting, non-blocking function*/
MPI_Request request;
MPI_Isend(..., &request);
MPI_Wait(&request, ...);
/*Example 2: Non-interrupting, non-blocking functions*/
MPI_Request request_dummy;
MPI_Isend(..., &request_dummy);
MPI_Request_free(&request_dummy);
/*Example 3: Single non-interrupting, non-blocking function*/
#define MPI_REQUEST_NULL null
MPI_Isend(..., MPI_REQUEST_NULL);
```

Fig. 2. A sketch of various non-blocking implementations. Example 1 and 2 describe how non-blocking functions are handled in typical message passing implementations. Example 3 showcases a minor modification on the MPI standard that provides increased performance for Class II interactions.

However, for Class II software-hardware interactions, request handles are often undesirable since they limit the performance of embedded processors. This arises from a typical use case of Class II interactions that has critical differences from the Class I interactions. The role of the embedded processor is to facilitate hardware engines with the appropriate data and not to complete computationally intensive tasks. In Class II interactions, non-blocking calls are extremely useful, not to overlap communication and computation, but to queue messages for the DMA hardware using a “fire-and-forget” paradigm. In this use case, the status of the message is not required and its mandatory implementation in non-blocking calls can greatly decrease performance. Request handles are serviced by interrupts, which require an expensive context switch on the embedded processor. Since the message status is not required, additional performance can be obtained by avoiding the interrupts used to update the message status.
In the MPI protocol, there is a provision that allows request handles to be freed to avoid throwing interrupts for the same performance benefit through the MPI_Request_free() function, shown in Fig. 2 – example 2. While this implementation works well for HPC cluster Class I interactions, a second function call still incurs significant overhead for an embedded processor and impedes Class II interaction performance.

For embedded MPI implementations, it is most efficient to design a single non-interrupting, non-blocking function that will not update the message status through interrupts. This can be easily implemented by modifying standard MPI non-blocking functions to accept a null pointer in place of the request handle. The non-blocking function checks the request handle and sets the interrupt for that particular message accordingly (Fig. 2 – example 3). This saves the overhead of the second function call for MPI_Request_free() while also ensuring that unnecessary interrupts are not thrown.

This implementation of a single non-interrupting, non-blocking MPI function does not comply with the MPI standard, but is a minor modification that greatly enhances the performance of embedded MPI Class II interactions.

D. Series of MPI messages – MPI_Coalesce()

The final consideration for Class II software-hardware interactions is the introduction of a new MPI function, called MPI_Coalesce(). In the typical embedded MPI use case, the processor is used for control and memory distribution via the ‘just-in-time’ message paradigm, where short messages with sufficient data are sent frequently to hardware engines. Applying the previously discussed optimizations and considerations, the embedded processor uses non-interrupting, non-blocking functions to queue numerous messages sequentially in the DMA hardware core while the compute engines are busy.

However, each function requires overhead since the return address and some registers are stored and later retrieved from the stack. Considering that a DMA-enabled message only requires the queuing of four words, stack accesses consume almost half of the time for each MPI function (Fig. 3).

**Fig. 3.** A time profile of the major components in a non-interrupting, non-blocking DMA function call. Obtained using a cycle accurate simulation of a PowerPC405 and memories, running at 300MHz and 100MHz, respectively.

MPI_Coalesce() is introduced as an alternative MPI function to transfer a series of messages while only incurring the overhead of a function call once. The MPI_Coalesce() uses a similar construct to other point-to-point MPI message functions – however, the function requires an array of arguments instead individual values. The function is implemented as a loop that iterates through each argument array and executes each function inline, as shown in Fig. 4.

The design of MPI_Coalesce() provides an effective trade-off between program size and performance for HPC embedded MPI implementations, where both properties are critical. One approach would be to inline every MPI function – although this eliminates all function call overhead, the program size grows severely. On the other hand, executing every MPI function reduces performance. MPI_Coalesce() provides an effective compromise where a single function call can be amortized across multiple, sequential MPI functions while only increasing the program code by a constant amount.

![Fig. 4](image.png)

**Fig. 4.** A sketch of the MPI_Coalesce() function.

IV. CLASS III: HARDWARE-BASED OPTIMIZATIONS

A. Background

This section describes a new message passing model that is well suited for Class III hardware-hardware interactions. Unlike the processor used in Class I and II interactions, hardware engines are capable of considerable parallelism and the standard message passing model is unable to fulfill the potential performance of these hardware engines.

In the standard software message passing model, each message and its corresponding function call is atomic. Thus, the program flow is quantized into indivisible units according to the messages. As a result, communication and computation cannot be executed simultaneously by a single processor.

However, unlike sequential processors, hardware compute engines can easily be designed with a significant degree of parallelism. Extending the message passing model so that hardware cores can receive, compute and send messages simultaneously greatly increases performance. This parallel use case of concurrent communication and computation is ubiquitous in hardware design.
B. Dataflow Message Passing Model

For hardware designers, problems are often stated in terms of dataflow – hardware is designed by considering datapaths and then building control logic to complement them. Computation is expressed as the movement of data through various circuits. This differs radically from the formal algorithmic approach used in software design, which defines a series of operations to be executed. The standard message passing is derived from the software model. Unfortunately, applying this message passing model to compute engines limits the hardware’s capabilities. A dataflow message passing model would better suit Class III hardware-hardware interactions.

In the standard message passing model, the atomicity of messages prevents concurrent communication and computation. Entire messages must be transmitted and processing cannot occur on the elements of partially transmitted messages. For Class III hardware-hardware design, rephrasing the message passing model so that each element in the message is treated as an independent component is more intuitive for hardware designers and provides increased performance.

A consequence of the dataflow message-passing model is that a single hardware MPI engine may represent multiple ranks simultaneously. For computation that requires multiple messages for execution, the parallelism of hardware designs permits the concurrent processing of messages to obtain maximum throughput. This multi-rank hardware model is analogous to a multicore HPC cluster node with multiple network interfaces using shared memory for intercore communication.

C. Case Study: Vector Addition

To illustrate the dataflow message passing model, a case study using a vector addition operation is presented. The computational engine must receive two messages from two different ranks each containing one vector, add the vectors in an element-wise fashion and the resulting vector is sent back to the two source ranks. The vectors are assumed to have \( n \) elements with a bit-width of \( w \), all operations require one cycle and there is no overhead to initialize message transfers.

```c
int v1[N], v2[N], v3[N];
/*Time Complexity*/
MPI_Recv(v1, N, MPI_INT, R1, ...);
// n stores
MPI_Recv(v2, N, MPI_INT, R2, ...);
// n stores
for (int i = 0; i < N; i++)
    v3[i] = v1[i] + v2[i];
// 2n loads, -- n add, n stores
MPI_Send(v3, N, MPI_INT, R1, ...);
// n loads
MPI_Send(v3, N, MPI_INT, R2, ...);
// n loads
```

Fig. 5. Vector addition using the typical message passing model.

For the traditional message passing model, the protocol outlined in Fig. 5 is executed and the architecture is described in Fig. 6.A.i. Although MPI functions, such as MPI_Allreduce(), may provide slightly better performance, basic MPI functions are used for illustrative purposes. This implementation results in a total time of \( 8n \) cycles (Fig. 6.A.ii) and requires at least \( 2nw \) memory units, if memory is reused destructively.

This implementation is ineffective in terms of performance and use of memory. Since the messages are atomic, execution of each message and computation is automatically serialized. However, the entire message is not required to begin computation – only one pair of elements from each message is required to produce one resulting element. Thus, if the granularity of the messages can be decomposed to operate on single elements, communication and computation can be done simultaneously.

Using the dataflow message passing model, a hardware design outlined in Fig. 6.B.i is used. This architecture allows the hardware to read one element from both ranks in one pipeline stage, complete the arithmetic in the second stage, and send the result in the final stage (Fig. 6.B.ii). This implementation results in a total time of \( n + 2 \) cycles and requires only \( 3w \) memory units (Fig. 6.B.iii).
It is important to note that the performance speed-up is not a result of adding the vectors in parallel, since a single adder is used in both cases. Instead, the dataflow message passing model provides a means to stream messages and process elements in a pipelined manner as opposed to being restricted to the coarse granularity of atomic messages.

As shown by this case study, the dataflow message passing model for Class III interactions provides a natural model for hardware designers to obtain further performance enhancements over the standard message passing model. Significant reductions in processing time and resource utilization can be obtained by decreasing the granularity of message operations without changing the message content or structure. Furthermore, this case study illustrates how a single hardware core represents multiple ranks for concurrent communication.

In addition, this dataflow message passing model illustrates how message passing models can be extended to allow for data streaming, where a continuous flow of data can be passed through a hardware engine. Using the available parallelism of hardware designs, embedded MPI implementations and compute engines can support full-duplex communication, allowing for simultaneous data transfer in both directions. This streaming extension drastically increases performance while also providing a natural model for hardware designers.

The dataflow model is an extension of the standard model that improves the performance of Class III hardware-hardware interactions. This model does not call for the modification of messages or how they are transferred, but instead reuses the existing framework. If the multi-rank extension is not used, the functional behaviour of the dataflow model is identical to the standard model and can be executed in software with standard MPI libraries. For multi-rank functional software simulations, a method to transfer information between ranks without messages, such as a shared memory system, is required.

V. HARDWARE DEBUGGING INTERFACES

This section describes how hardware cores can be used to debug embedded MPI implementations. With a focus on maintaining code compatibility, the first step in debugging embedded MPI applications is to use traditional MPI debugging methods on a software-only platform. However, porting the design to embedded platforms can result in unexpected issues such as improper on-chip network setup or message passing flaws in the compute engines. The ability to debug hardware MPI designs is further complicated by the limited visibility – hardware units often lack popular analysis tools such as tracing debuggers or even standard output.

Rather than analyzing the system solely through the endpoints as one would do in a software implementation, an effective method of debugging embedded MPI designs is to augment the communication network with smart hardware cores that allow messages to be traced through the communication links. Thus, two hardware network debugging interfaces were developed to help debug hardware MPI designs.

A. Tee Cores

The first hardware debugging interface is the Tee core. The typical embedded communication network consists of point-to-point FIFOs connecting each processing node. Tee cores replace the FIFO links with hardware junctions that duplicate the transmitted data on the original link through a secondary link, which acts as a queue that stores the last set of transmitted data. The Tee cores are transparent and do not affect the original operation of the communication network.

The secondary link can then be connected to a debugging interface, such as a processor. This allows the processor to actively snoop messages in the network and report the status of the application. Large queues can be instantiated to ensure that no data loss.

Tee cores are analogous in the HPC cluster paradigm to having a physically wired junction in the interconnects that relays information and stores the last data in a queue for later retrieval. Although debugging at the data link layer may seem like a low level approach for MPI implementations, the relatively simple networks and communication protocols in embedded applications as well as the limited visibility of hardware endpoints makes this an effective approach.

B. Message Watchdog Timers

The next debugging interface is watchdog timers that are integrated with the MPI implementation source code. Unlike HPC clusters, embedded systems are particularly difficult to debug since they lack a universal input that allows the user to interrupt and regain control of the system. In typical embedded systems, an unresponsive system cannot be recovered.

The design of message watchdog timers helps to resolve this problem. These timers actively snoop the communication network for activity in a transparent manner. If there is no activity on the network after a parameterizable amount of time, the watchdog timer interrupts the associated processor and provides a method to recover locked systems.

The message watchdog timers are particularly effective when coupled with the Tee cores to create a post-mortem debugging interface. Once a system is locked, the message watchdog timer interrupts the processor. The processor can then retrieve the last set of messages from the Tee cores, providing a terse debugging report. The ability of both interfaces to work transparently ensures that accurate debugging information is provided to the user.

VI. IMPLEMENTATION RESULTS AND ANALYSIS

To illustrate the effect of these optimizations, an embedded MPI research application that utilizes these modifications is analyzed. The research project is a large-scale FPGA accelerator designed to increase the performance of Restricted Boltzmann Machines (RBMs) [14], a popular neural network. This implementation uses the message-passing paradigm to create a large-scale hardware design.

The design is implemented on the Berkeley Emulation Engine 2 (BEE2) board [12], which has five Xilinx I-PRO XC2VP70 FGPAes connected in a communication mesh with 6-cycle latency and a bandwidth of 1.73GB/s between pairs of computing FGPAes. The PowerPC runs in standalone mode at 300MHz while the compute engines operate at 100MHz. In this example, a 256×256 node RBM is instantiated where each FPGA computes a 128×128 portion of the larger network.

The design uses an MPI architecture shown in Fig 7. The system is fairly complex, consisting of one PowerPC405
processor and 12 compute engines spread across four FPGAs resulting in 21 MPI ranks. Both classes of embedded MPI interactions are used: the processor is responsible for control and memory distribution (Class II), while the hardware units use the dataflow message passing model (Class III).

A. Class II interactions

Neural networks operate by processing data vectors through automated rules which adjust the appropriate learning parameters. In our implementation, the processor retrieves the data vectors from memory and sends them to the compute engines via Class II interactions, summarized in Table II. Once the compute engines receive the data vectors, they only communicate with other hardware engines until the next set of data vectors is required. This type of operation adheres to the 'just-in-time', producer-consumer embedded message model where the processor sends just enough information for one iteration of computation.

### Table II

**CLASS II MESSAGES FOR ONE ITERATION OF RBM COMPUTATION**

<table>
<thead>
<tr>
<th>#</th>
<th>Src</th>
<th>Dest.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R0</td>
<td>R1</td>
<td>0 words</td>
</tr>
<tr>
<td>2</td>
<td>R0</td>
<td>R1</td>
<td>3 words</td>
</tr>
<tr>
<td>3</td>
<td>R0</td>
<td>R6</td>
<td>0 words</td>
</tr>
<tr>
<td>4</td>
<td>R0</td>
<td>R6</td>
<td>3 words</td>
</tr>
<tr>
<td>5</td>
<td>R0</td>
<td>R11</td>
<td>0 words</td>
</tr>
<tr>
<td>6</td>
<td>R0</td>
<td>R11</td>
<td>3 words</td>
</tr>
<tr>
<td>7</td>
<td>R0</td>
<td>R16</td>
<td>0 words</td>
</tr>
<tr>
<td>8</td>
<td>R0</td>
<td>R16</td>
<td>3 words</td>
</tr>
<tr>
<td>9</td>
<td>R0</td>
<td>R1</td>
<td>4 words</td>
</tr>
<tr>
<td>10</td>
<td>R0</td>
<td>R6</td>
<td>4 words</td>
</tr>
<tr>
<td>11</td>
<td>R0</td>
<td>R11</td>
<td>4 words</td>
</tr>
<tr>
<td>12</td>
<td>R0</td>
<td>R16</td>
<td>4 words</td>
</tr>
</tbody>
</table>

The following series of incremental, embedded processor-based MPI configurations is analyzed:

- **A) Processor I/O** – This baseline platform has MPI messages executed through processor memory copies in software. The processor is connected to the communication network through a custom Device Control Register to FSL bridge (DCR2FSL).

- **B) DMA with blocking messages** – This platform uses the PLB_MPE core to implement DMA messages. Only blocking messages are used.

- **B + C) DMA with non-interrupting, non-blocking messages** – This platform uses non-blocking, non-interrupting messages to enqueue messages while the hardware engine is busy. The last message is left as a blocking call to synchronize the processing iterations.

- **B + C + D) DMA with MPI_Coalesce()** – This platform will use MPI_Coalesce() to integrate all of the messages in the previous configuration into a single function call.

The timing diagram of each of the implementations is analyzed in Fig. 8 and the total speed-up of using these messages is shown in Table III. Compared to the processor I/O implementation, the PLB_MPE DMA hardware core using MPI_Coalesce() is able to obtain a 5-fold speed-up. This significant increase in performance was a result of minor modifications that better suit the embedded FPGA environment.

### Table III

**IMPLEMENTATION STATISTICS**

<table>
<thead>
<tr>
<th>Platform</th>
<th>HW Idle Time [ns]</th>
<th>Iteration Period [ns]</th>
<th>Accumulated Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>73,580</td>
<td>85,200</td>
<td>–</td>
</tr>
<tr>
<td>B</td>
<td>25,240</td>
<td>36,560</td>
<td>2.33x</td>
</tr>
<tr>
<td>B + C</td>
<td>10,320</td>
<td>21,640</td>
<td>3.94x</td>
</tr>
<tr>
<td>B + C + D</td>
<td>4,680</td>
<td>16,000</td>
<td>5.32x</td>
</tr>
</tbody>
</table>

B. Class III optimizations

Rather than quantifying the clear performance benefits of using the dataflow model; the following analysis is an illustration of how the model can be used effectively in a real application. The FPGA RBM design [14] uses combinations of three hardware engines: the Restricted Boltzmann Machine Core (RBMC), the Node Select Core (NSC) and the Energy Accumulator Core (EAC). As part of its processing, the RBMC generates a vector of data, called energies, from the node...
states. In a complimentary fashion, the NSC uses the energies to determine node states. In a single FPGA implementation, the two cores are connected so that energies and node states are generated in a cyclical manner (Fig. 9.a).

However, modifications to the architecture are required to scale the design across multiple FPGAs. To preserve proper functionality, the energies from multiple RBMCs must be combined before the NSC processes them. Likewise, the node states from the NSC must be duplicated and sent to each RBMC. The EAC provides this functionality and effectively abstracts the operation of the RBMC and NSC (Fig. 9.b).

The EAC takes advantage of both the multi-rank and streaming paradigms in the dataflow message-passing model. At a lower level, the EAC begins by initiating messages with both the RBMCs and NSCs. Once each of the compute engines is ready to transmit energies and node states, the EAC then streams data bidirectionally through its compute engine using a pipelined datapath. This implementation of the dataflow message passing model provides a low overhead communication protocol that provides hardware designers a natural method to build compute engines. Furthermore, adhering to the message passing model provides a top level modular approach to constructing hardware for large-scale designs.

a) Single FPGA Implementation

```
   R1         R2
    \   /     / \   \
     \ /     /   \
      / \     /     \
 RBMC      EAC     NSC

b) Multi-FPGA Implementation

```

Fig. 9. A schematic of single FPGA RBM implementation (Part a), and it is expanded using the EAC and the dataflow message passing model (Part b).

VII. CONCLUSION AND FUTURE WORK

This paper describes several challenges and solutions in designing an MPI implementation that is suitable for embedded FPGA applications.

For Class II interactions, several improvements were introduced to accelerate producer-consumer use cases; the PLB_MPE allows processors to offload memory transactions to a DMA engine with MPI functionality; a single non-interrupting, non-blocking MPI function provided fast messages enqueuing; and MPI_Coalesce() was proposed to reduce the overhead for enqueuing a series of sequential messages. These improvements resulted in a speed-up of 5-fold compared to a software-only embedded MPI implementation.

For Class III interactions, a dataflow message passing model is presented that allows compute engines to communicate and compute simultaneously. The use of multi-rank hardware promotes streaming applications, providing a natural method for hardware designers to build high-performance MPI engines.

To aid in debugging hardware MPI designs, two hardware cores, Tee cores and message watchdog timers, are described. Tee cores provide users with a method to transparently snoop the messages on communication network while the message watchdog timers help to recover debugging information from an unresponsive embedded processor.

The proposals in this paper are designed as an extension of the TMD-MPI framework to improve its efficiency and performance. Incremental improvements were used in an attempt to maintain compatibility. Although some of the improvements do not directly comply with the MPI standard, all of the extended semantics can be reduced to legitimate MPI code. This suggests that our proposed improvements are reaching the limit where compatibility will be compromised for performance. Thus, if MPI intends to be a universal message passing model, this work clearly shows the need to consider the properties and trade-offs of various platforms and extend the current standard.

Future work includes a deeper investigation of the dataflow model and how various types of fine grain parallelism can be exploited within the message passing paradigm.

ACKNOWLEDGEMENT

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project.

REFERENCES


