An Extended GASNet API for PGAS Programming on a Zynq SoC Cluster

by

Sanket Pandit

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Rogers Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Sanket Pandit
Abstract

An Extended GASNet API for PGAS Programming on a Zynq SoC Cluster

Sanket Pandit

Master of Applied Science

Graduate Department of Rogers Department of Electrical and Computer Engineering

University of Toronto

2016

This work presents a dynamic development flow to integrate FPGA accelerators into software applications for a Zynq SoC cluster. We provide a high-level library, THe_GASNet Extended API, based on the Partitioned Global Address Space (PGAS) programming model for remote memory operations. The extended API is built using the THe_GASNet Core API that implements a global address space view of the memory but provides distinction between local and remote data accesses. The extended API provides an expressive way to implement parallel applications on the x86-ARM-FPGA heterogeneous systems. For the benchmark, we implemented the Jacobi iterative method on the cluster.

The hardware implementation of the Jacobi method performs 2.78-fold faster than the software implementation on the i5 processor for a 4096x4096 surface. Compared to the core API, the extended API performs slightly worse for all the data sizes. However, the trade-off of the reduced performance is increased ease of programming.
Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Paul Chow for providing me the opportunity to work with him, for showing unwavering confidence in my abilities, for giving me the independence to take this work in any direction and allowing me to learn from my mistakes, for patiently helping me in my writing, and for providing motivation at very trying times.

This thesis would have not been possible without the groundwork laid by Ruediger Willenberg, who also provided extensive support throughout my research. I am very grateful for having such an excellent mentor and a friend. I would also like to thank Charles Lo for providing very helpful advice throughout the research which has greatly affected the results of this work.

I started my masters with few undergraduate friends whom I have gotten to know over the years: Nadeesha Amarasinghe, Peter Chang, Hassan Farooq, Yasser Khan, Wen Bo Li, Rafid Mahmood, Namal Rajatheva. Thank you for making the university one of the most cherishable experience of my life.

Throughout the graduate school, I met many excellent people and some I have gotten to know as friends: Mario Badr, Fernando Martin del Campo, Jasmina Capalija, Joy Chen, Xander Chin, Ehsan Ghasemi, Keiming Kwong, Jin Hee Kim, Charles Lo, Vincent Mirian, Angelia Tian, Justin Tai, Naif Tarafdar, Cynthia Shin, Cathy Zhu and many others. Thank you for making my journey enjoyable and giving me something to look forward to every day at work.

Finally, I would like to thank my parents, and my brother for their incredible love, support, and encouragement throughout my studies. I am forever grateful to my parents who uprooted themselves to provide us the opportunities that otherwise would not have been possible.
# Contents

1 Introduction 1

1.1 Motivation 1

1.2 Research Contributions 2

1.3 Thesis Organization 2

2 Background 4

2.1 Memory Models 4

2.1.1 Shared Memory 4

2.1.2 Distributed Memory 7

2.1.3 Partitioned Global Address Space (PGAS) 9

2.2 Active Messages 12

2.3 Related Work 12

2.3.1 GASNet Library 12

2.3.2 SHMEM/openshmem 13

2.3.3 Unified Parallel C (UPC) 14

2.3.4 CoRAM 14

2.3.5 SHMEM+ 15

2.3.6 TMD-MPI 15

2.3.7 THe GASNet Core API 16

3 Hardware Implementation of the THe_GASNet Extended API 21

3.1 System Overview 21

3.2 Off-Chip Network 25

3.3 On-Chip Network 26

3.4 Datapath Example 27
### 3.5 DMA between PS and PL


### 3.6 The GASNet Supporting IPs

- **3.6.1 The Global Address Space Core (GASCore)**
- **3.6.2 Extended Programmable Active Message Sequencer (xPAMS)**

### 4 The GASNet Extended API

- **4.1 The Programming Model**
- **4.2 The GASNet Core API Overview**
  - **4.2.1 The GASNet Core API Communication Functions**
  - **4.2.2 The GASNet Core API Runtime Structure**
- **4.3 The GASNet Extended API Overview**
  - **4.3.1 Handles**
  - **4.3.2 The GASNet Extended API Communication Functions**
- **4.4 PAMS Control Interface**
- **4.5 Point-to-Point Transfer Operation**

### 5 Results and Analysis

- **5.1 Microbenchmarks**
  - **5.1.1 Latency**
  - **5.1.2 Throughput of the AXI-FSL DMA**
  - **5.1.3 Throughput of the Datapath**
- **5.2 Heat Transfer Equation Case Study**
  - **5.2.1 Jacobi Software Implementation**
  - **5.2.2 Jacobi Hardware Implementation**
  - **5.2.3 Code Comparison**
  - **5.2.4 Experimental Setup**
  - **5.2.5 System Performance**
  - **5.2.6 Resource Utilization**

### 6 Conclusions

- **6.1 Future Work**

### Appendices
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The GASNet Core Functions</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>The GASNet Extended Functions</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>PAMS Control Interface</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Latency of on-chip communication and off-chip communication</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Utilization of the PL Region in the Zynq SoC for Two Hardware Nodes</td>
<td>71</td>
</tr>
<tr>
<td>A.1</td>
<td>Active Message parameters - Standard types</td>
<td>76</td>
</tr>
<tr>
<td>A.2</td>
<td>Active Message parameters - Strided Long type</td>
<td>77</td>
</tr>
<tr>
<td>A.3</td>
<td>Active Message request parameters - Standard types</td>
<td>78</td>
</tr>
<tr>
<td>A.4</td>
<td>Active Message request parameters - Strided Long type</td>
<td>78</td>
</tr>
<tr>
<td>A.5</td>
<td>Handler function call parameters - Standard types</td>
<td>78</td>
</tr>
<tr>
<td>A.6</td>
<td>Handler function call parameters - Strided Long type</td>
<td>79</td>
</tr>
<tr>
<td>C.1</td>
<td>DMA Write Throughput and Latency</td>
<td>83</td>
</tr>
<tr>
<td>C.2</td>
<td>DMA Read Throughput and Latency</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Shared memory after running openMP program ........................................ 6
2.2 Distributed memory after running MPI program ........................................ 9
2.3 PGAS memory after running SHMEM program ......................................... 11
2.4 System Diagram of GASNet API Layers .................................................. 13
2.5 System Diagram of THe_GASNet Core API Layers highlighted in Red Outline . 16
2.6 MapleHoney Reconfigurable Computing Cluster (Source: R.Willenberg) .......... 17
2.7 MapleHoney: internal design of a single FPGA (Source: R.Willenberg) .......... 18
2.8 System Diagram of the THe_GASNet Extended API Layers highlighted in Red Outline . 20
3.1 ZedBoard Cluster: inside the PL ............................................................. 22
3.2 DDR memory partition ................................................................. 24
3.3 ZedBoard Cluster ................................................................. 24
3.4 On-Chip Network implemented in Programmable Logic ................................ 26
3.5 Datapath Example of an Off-chip Node-to-Node Transfer ............................ 27
3.6 DMA module to implement hardware-software communication ................. 28
3.7 Hardware Node Implementations ....................................................... 30
3.8 Extended Programmable Active Message Sequencer .................................. 33
3.9 CodeRAM Memory Map .............................................................. 36
4.1 Point-to-Point Memory Transfer Example .............................................. 47
5.1 Throughput of the AXI-FSL DMA ....................................................... 53
5.2 Throughput of the software-to-hardware datapath ...................................... 55
5.3 2D Surface Paritioning (Source: R.Willenberg) ...................................... 57
5.4 Node-to-Node Communication between Iterations (Source: R.Willenberg) .... 57
5.5 Surface Length vs. Runtime Results for 1024 iterations of the Jacobi Method ... 66
5.6 Breakdown of the Runtime of Jacobi method for 1024 Iterations for the ARM Cluster,
the ARM-FPGA Cluster, and the Intel i5 Processor on a 4096x4096 Surface ........... 66

5.7 Comparison of the Runtime of 1024 Iterations of the Jacobi Method ................. 68

5.8 Performance of the PAMS vs. the xPAMS for 1024 Iterations of the Jacobi Method
using Eight Hardware Nodes ........................................................................... 69

5.9 Performance of The GASNet Extended API vs. The GASNet Core API in Software . 70

B.1 Low-level PAMS/xPAMS instruction set ....................................................... 81
Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) allow rapid hardware design compared to Application Specific Integrated Circuits (ASIC) making them suitable for computing applications because they can be easily reprogrammed. Despite relatively low clock frequencies, FPGAs provide a significant performance boost for certain applications compared to the x86 or ARM architecture. Naturally, FPGAs should be a good fit for High Performance Computing (HPC) systems where performance is a primary goal. Even with high-level synthesis (HLS) options available from both major FPGA vendors, adoption of FPGAs in HPC systems has been limited. The slow adoption can be attributed to two major factors: the learning curve involved in bringing the software programmers closer to the hardware design when using HLS [1], and the longer implementation times for FPGAs compared to the software compilation times.

The High-Performance Re-configurable Computing group at University of Toronto is focusing on creating a development flow to allow programmers to implement distributed parallel programming applications with the Toronto Heterogeneous GASnet Core API (THeGASnet Core API) using the Partitioned Global Address Space (PGAS) programming model [2]. THe_GASNet Core API provides low-level communication support to the accelerators/soft-processors implemented on the FPGA. While the core library implements the message passing interface, it leaves the implementation of the handler functions, message type, implementing memory fetch functions to the users. The limited functionality provided by the library hinders the ability to port other applications and PGAS runtime libraries to the existing environment. A higher level abstraction is required which is the focus of this work.
1.2 Research Contributions

The goal of our research is to test the feasibility of FPGAs in a PGAS programming model to accelerate HPC applications. To this end, we have implemented a low-overhead, high-level communications library that can be leveraged by a PGAS runtime library to provide communication between the computation nodes located on FPGAs, on ARM processors, or on x86 processors. A high-level library also allows us to port applications provided by other PGAS libraries to our Zynq SoC cluster furthe...
experiments that characterized the system to identify the bottlenecks. The chapter later discusses the heat transfer application that we implemented as a use case to assess the performance of the cluster against an x86 processor. Finally, Chapter 6 concludes and discusses possible ways to improve the performance of the architecture.
Chapter 2

Background

This chapter reviews some of the key concepts that underlie our work. This chapter begins by introducing memory models that are commonly used in modern multi-processing systems. We describe how these models differ from each other by providing example code emphasizing the differences and discussing challenges associated with the programming models. We then present the message-passing paradigm that is used to implement data transfers in our system. The final section gives a brief overview of the ongoing research in the field of programming models and their implementation on the FPGAs.

2.1 Memory Models

A memory model presents an abstracted view of the memory to an application programmer. This allows the programmer to use memory that may or may not be physically located near the device as part of the device, or it may allow the programmer to treat one physical memory as distributed among the processors. Choosing the right memory model can reduce the latency in the system and promote scalability. Most modern multi-processing systems make a choice between implementing one of the three major memory models: shared memory, distributed memory, and distributed shared memory.

2.1.1 Shared Memory

Shared memory refers to a programming model that provides access to all the memory under one global address space. A global address space means that all the available memory has a fixed address, and a parallel processor can access any memory directly by referencing that fixed address. The shared memory model can be further divided based on how the architecture provides accessibility:
Uniform Memory Access (UMA) : refers to memory models where access to the memory is accomplished through a central interconnect unit (software or hardware) that implements the global address space. Since all the memory requests have to go through the central interconnect, this gives equal latency and bandwidth to all the processors.

Non-Uniform Memory Access (NUMA) : refers to memory models where each processor has an attached memory, however, using a central interconnect all the memory in the system is mapped to one global address space. At the high level, processors can access the remote memory similar to local memory, however, the performance of a remote access will differ compared to the local memory access.

OpenMP is a shared memory multi-processing programming model developed for C, C++, and Fortran. Listing 2.1 implements an example application for OpenMP that transfers “Hello” and “Good-Bye” messages between the parallel nodes. The “Hello” message is created by Node 0 and “GoodBye” messages are created by all the other nodes. The application transfers the “Hello” message from Node 0 to all the other nodes and transfers the “GoodBye” messages from all the other nodes to Node 0.

In OpenMP, the transferring of messages is achieved using shared memory. Lines 7-10 generate shared memory buffers called hellobuf, final_hellobuf, idbuf and byebuf to hold the “Hello” and “GoodBye” strings respectively. The program starts with a single master thread until the pragma at line 11 forks slave threads. The number of slave threads is determined by the num_threads() parame-
Figure 2.1: Shared memory after running openMP program
ter, which is set to four. A single copy of `hellobuf`, `final_hellobuf`, `idbuf`, and `byebuf` is shared between all the slave threads. Since all the threads have access to the buffers, we can copy the “Hello” string generated by Node 0 to the shared buffer. The barrier pragma on line 16 blocks all the threads until Node 0 has finished copying the “Hello” message to the shared buffer. Other nodes can then read the message in the shared buffer region. Similarly, we can store all the “GoodBye” strings created by other nodes to the `byebuf` buffer, which is accessible to the Node 0. Figure 2.1 shows us the memory contents once the application has finished running. All the nodes in the system can read each other’s messages by simply accessing the location where the messages are stored.

The OpenMP example provides us with the key benefit of the shared memory model: it is easy to program since all the threads reference the same memory. While the shared memory model provides easy to reference memory, both UMA and NUMA models presented above use a central interconnect unit to provide the global address space. The interconnect performance can degrade if we increase the number of parallel nodes. It also leads to data contention when all the nodes are accessing the same data. Therefore, while easy to program, shared memory models do not scale very well.

### 2.1.2 Distributed Memory

The distributed memory model provides a private address space to all the parallel nodes. While aware of being a part of a parallel system, nodes cannot directly access remote memory. Communication between memories is done by sending messages between nodes. Since a node sending a message cannot access the remote memory, the destination node must also be involved in passing a message leading to a two-sided communication.

The Message Passing Interface (MPI) standard \[5\] has become the de-facto standard for implementing message passing for distributed systems. MPI uses `MPI_Send()` and `MPI_Recv()` functions to implement the two-sided communication. Listing 2.2 shows the example code that implements the Hello-Goodbye application mentioned in the previous section using MPI. While all the nodes in the system create the `hellobuf`, `src_byebuff`, and `dest_byebuf` buffers, all the nodes have a private copy in their address space. When Node 0 initializes the `hellobuf` with the message in line 23, only the private address space of Node 0 is modified. The Node 0 must send the message through `MPI_Send()` as shown in line 25 for other nodes to be able to read the message. Meanwhile, the other nodes are blocked at line 39 at the `MPI_Recv()` function waiting for the message. The `MPI_Recv()` function not only blocks the message, but it also passes the buffer address in the receiver node’s private memory where the incoming message is to be stored. In our example, the `MPI_Recv()` function at line 39 uses `hellobuf`,
but it can use any memory location to store the incoming message. Similarly, the “Goodbye” message is created by all the other nodes in their private memories in line 41 and passed to Node 0 through the send-receive mechanism. Figure 2.2 shows the contents of the memories once the application has been run. While all the nodes create the same buffers, they can be located anywhere in the physical address space of each processor.

As the example code shows, two-sided communication allows greater control over the program flow through the MPI_Send() and the MPI_Recv() functions. However, the need to manage the communication along with the computation code makes the application hard to program. As the application requires more complex message passing, the programming can become significantly more complicated.

```c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define THREADS 4
#define TAG 0

int main(int argc, char *argv[]) {
    char hellobuf[32];
    char src_byebuf[32];
    char dest_byebuf[THREADS][32];
    int numprocs, myid, i;
    MPI_Status stat;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if(myid == 0) {
        for(i=1;i<numprocs;i++) {
            //Initialize the hellobuff with the "Hello" message
            sprintf(hellobuf, "Hello %d!\n", i);
            //Send "Hello" message to all the nodes
            MPI_Send(hellobuf, 32, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
        }
        sprintf(hellobuf, "Hello 0!\n");
        for(i=1;i<numprocs;i++) {
            //Wait for "Goodbye" message from all the nodes
            MPI_Recv(dest_byebuf[i], 32, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
        }
    } else {
        //Copy "Hello" messages from Node0 to the local hellobuff
        MPI_Recv(hellobuf, 32, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
        //Initialize the src_byebuff with the "Goodbye" message
        sprintf(src_byebuf, "Goodbye %d\n", myid);
        //Send "Goodbye" message to Node0
        MPI_Send(src_byebuf, 32, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Listing 2.2: Hello-Goodbye Application using MPI
2.1.3 Partitioned Global Address Space (PGAS)

The PGAS memory model, also known as the Distributed Shared Memory, provides us with the best of both worlds. It provides the data locality found in distributed memory, and provides an easy to reference global address space found in shared memory. While the memory is referenced globally, PGAS partitions the memory such that each partition has an affinity to a node. Since a node has access to the global memory through the global address space, it can perform one-sided communication for data transfers. One-sided communication allows a process to access memory located in another node without requiring any actions from the destination node. Note that the destination node may be involved in providing support for the communication, but a programmer does not have to account for incoming communications in the application code. The handling of incoming requests is done implicitly by the communication library, or handler functions. Whereas in the two-sided communication described in the previous section, the programmer must implement receive functionality explicitly to handle incoming requests. One-sided communication removes the explicit participation of the destination node in the communication, thereby increasing the possibility of overlapping communication and computation.

SHMEM/openshmem [6] is one of the popular libraries used to implement the PGAS programming model for C/C++ applications. SHMEM achieves parallelism through the Single Program stream, Multiple Data stream (SPMD) technique. SPMD refers to the type of parallel systems where all the nodes run the same program, however they work on different sections of the data. Listing 2.3 shows the example of the Hello-GoodBye application implemented using the SHMEM library. Similar to OpenMP, we allocate memories that are used as buffers using the shmalloc() function. However, the shmalloc() function allocates the buffer in other memory partitions symmetrically. Symmetric
#include <mpp/shmem.h>
#include <stdio.h>

main()
{
    int me, npes;
    int i=0;
    start_pes(0);
    npes = _num_pes();
    me = _my_pe();

    char* src_hellobuf;
    char* dest_hellobuf;
    char** dest_byebuf;
    char* src_byebuf;

    //Allocate the buffers symmetrically across all nodes
    src_hellobuf = (char*)shmalloc(sizeof(char)*32);
    src_byebuf = (char*)shmalloc(sizeof(char)*32);
    dest_hellobuf = (char*)shmalloc(sizeof(char)*32);
    dest_byebuf = (char**)shmalloc(sizeof(char*)*npes);

    for(i=0;i<npes;i++)
        dest_byebuf[i]=(char*)shmalloc(sizeof(char)*32);

    if (shmem_my_pe() == 0) {
        for(i=1; i<npes; i++) {
            //Initialize the buffer with the "Hello" message
            sprintf(src_hellobuf, "Hello %d!\n", i);
            //Put the message into remote memory's destination buffer
            shmem_int_put(dest_hellobuf, src_hellobuf, 1, i);
        }
        sprintf(src_hellobuf, "Hello 0!\n");
    } else {
        //Initialize the buffer with the "Goodbye" message
        sprintf(src_byebuf,"Goodbye %d!\n", me);
        shmem_int_put(dest_byebuf[me], src_byebuf, 1, 0);
    }
    shmem_finalize();
}
allocation means that relative to the starting address of the partition, buffers are located at the same offset. Therefore, relative addressing can be used to copy the data in a remote memory’s buffer. In line 28, Node 0 generates the “Hello” message in its local memory first, and then using `shmem_int_put()` passes the data to the buffer of the destination node. Similarly, “GoodBye” messages are transferred to the `dest_hellobuf` buffer that is accessed by Node 0.

While the programming approach is similar to OpenMP, the memory region in the PGAS programming model is organized differently as shown in Figure 2.3. Unlike the shared memory model, PGAS partitions the global address space for each node shown by the dashed line. Since the PGAS programming model has a global address space, symmetric memory can be created on multiple partitions. Symmetric memory permits the ability to copy data at a remote memory’s buffer region using relative addressing, making all the local partitions transparent. This allows us the scalability of the distributed memory systems, at the same time providing an easy to use programming interface.

While MPI-2 has implemented one-sided communication, the implementation support is still limited.
2.2 Active Messages

One-sided communication that is discussed in Section 2.1.3 can be implemented using Active Messages (AM). AMs were developed by Eicken et al. [7] as an asynchronous communication mechanism. “Active” in AM refers to the fact that the messages provide the information for further processing, whereas passive messages only contain the data and processing is done beforehand, as for example with MPI Send() and MPI Recv(). An AM carries information about its handler function in the header of the message. When the AM reaches the destination, the handler function is executed to remove the AM from the network. The active message can interrupt the processor to invoke a handler function to transfer that specific message to a buffer. However, in some implementations, handler functions are run by a separate handler thread without the involvement of the main thread. Since the source node needs to be aware of all the handlers available at the destination node, the same code must be loaded on all the nodes, making AMs an ideal choice for the PGAS SPMD model.

One major advantage for AMs is that it reduces the communication time compared to the send-receive method. In the send-receive method, before the actual data is sent, it passes two messages over the network (send and receive). With the AM, only one message needs to be passed, thereby making more efficient use of the network. Since there is no synchronization needed between the nodes for communicating data, it also makes programming an application much easier.

2.3 Related Work

This section presents the current research in the field of PGAS programming model by introducing various libraries and languages. It also presents research that implements programming models to the FPGAs.

2.3.1 GASNet Library

GASNet [8] is a network-independent and language-independent low-level networking library developed at U.C. Berkeley and Lawrence Berkeley National Lab. GASNet provides a high-performance communication interface for implementing PGAS languages [9] as shown in Figure 2.4. GASNet is developed as a compilation target so performance is the key driver instead of readability. The GASNet library can be further divided into the GASNet Core API and the GASNet Extended API. The core API provides a bare-essential library that implements the active message functionality over a network. The core API uses one-sided active message communication for passing data between the parallel nodes. The core
API is not network-independent, thus it has to be implemented for each network interface. Therefore, keeping the GASNet Core API simpler enhances portability, since only the core API library needs to be ported. The extended API provides medium to high-level memory transfer functions for remote memory operations. A network-independent extended API is implemented purely in terms of the core API functions. While most of the communication needs are taken care of through the extended API, the runtime library has the option to implement language-specific, network-specific, high-performance features using the core API directly. GASNet has been used to implement languages and libraries such as UPC [10], Co-Array Fortran [11], SHMEM [6], Cray Chapel [12], and Titanium [13].

2.3.2 SHMEM/openshmem

SHMEM [6] is a family of high-level PGAS libraries that was initially developed for providing one-sided communication messages supported by Cray PVP systems, Cray MPP systems, and the Silicon Graphics systems. However, the library was not meant to be implemented as a standard protocol and did not extend support to other systems. This lead to an effort to create a standard API for SHMEM libraries called openshmem driven primarily by the Extreme Scale Systems Center (ESSC) at ORNL and the University of Houston. This library has played a key role in determining what support is expected from a high-level PGAS library.
2.3.3 Unified Parallel C (UPC)

Unlike GASNet and SHMEM/openshmem, which are both communication libraries, UPC is an extension to the C programming language to provide high-performance PGAS implementations on large-scale parallel systems [10]. The UPC runtime system is implemented on top of the GASNet API to provide low-overhead one-sided communication. UPC divides the global address space further into private memory and shared memory. Regular variables and objects that are initialized during runtime are located in the private memory, whereas variables with the shared type are initialized in the shared regions. The variables available in the shared region are directly available to any processor in the system through a pointer-to-shared type. Another key feature of language extensions or languages such as UPC or Chapel is that arrays can be initialized automatically over the parallel system. Distribution of that array over the parallel system can be specified by the program for cyclic or block distribution among the processors. This significantly eases the programming for PGAS languages.

UPC++ [14] is being developed to provide an object-oriented PGAS programming model using C++. UPC++ still uses the GASNet API as a communication layer between the parallel nodes. UPC++ is used to analyze the performance of the PGAS library against MPI for a hydrodynamics operation that describes the motion of materials relative to each other when a force is applied. The 3D region for the hydrodynamics is divided among the parallel nodes where each node needs to communicate with 26 neighbors in the 3D region. The MPI implementation uses non-blocking send-receive functions, whereas one-sided communications are used for UPC++ communication. For the largest experiment run with 32K parallel nodes, the UPC++ implementation performs 10% faster than the MPI implementation. While in this case the performance of a PGAS library is faster than MPI, there are experiments where MPI performs faster than the PGAS library [15]. It is important to remember that the goal of the PGAS library is to provide an easy to program library that can use both shared memory, and distributed memory concepts, at the same time maintaining reasonable performance.

While there are many languages that are being developed for the PGAS programming model such as UPC, GASNet, SHMEM, Chapel, etc. very few of these languages/libraries have been implemented on FPGAs. Three key libraries that have influenced our work are: SHMEM+ [16], TMD-MPI [17], and the THe_GASNet Core API [2].

2.3.4 CoRAM

CoRAM [18] is one of the more recognized works that provides a way to abstract away the memory management when porting an application to the FPGA. The user can attach the processing elements to
a CoRAM block that performs memory transfers to or from an external shared memory. The CoRAM block is controlled by a software control thread using *control actions* that are written in C, which is later transformed to a Finite State Machine (FSM). The key focus of this work is to provide the processing elements a standardized memory control module that is controlled by the *control actions*; it does not focus on providing a standard programming model to the user. A change in communication or memory access still requires regenerating the bitstream, which our work avoids by using a generalized FSM.

### 2.3.5 SHMEM+

SHMEM+ [16] extends the SHMEM communication library to enable coordination between FPGAs and x86 processors in a high-performance re-configurable system. However, the majority of the functionality for SHMEM+ was provided through a software library for the x86 processors, and the FPGA was designed to only respond to transfer requests. A processing element for SHMEM+ includes an FPGA along with an x86 processor where communication between the x86 processor and the FPGA is accomplished using a vendor specific API, and SHMEM+ is implemented for x86 to x86 communication. One major drawback of the system is that communication cannot be initiated from the FPGA, since the FPGAs are programmed as slave accelerators, rather than more general model where the FPGAs are peers to the x86 processors. Therefore, the software program has to be involved in implementing low-level communication between FPGAs, which makes programming with large scale FPGA systems difficult.

### 2.3.6 TMD-MPI

TMD-MPI [17] is an implementation of the MPI standard for High Performance Reconfigurable Computers (HPRCs) that include multi-FPGA configurable systems with embedded processors, and x86 processors. Unlike the SHMEM+ implementation where the x86 processors are responsible for the communication, and the FPGAs are merely used as computation slaves, TMD-MPI implements the MPI standard as a uniform communication layer between the FPGAs and the x86 processor allowing the FPGAs to directly participate in the communication. However, the project has the same drawbacks as the MPI example shown in Section 2.1.2 where it gets difficult manage communication between individual components as the number of processing units increase because of the *two-sided communication*, and the need to allocate buffers for the message passing.


2.3.7 THe_GASNet Core API

The THe_GASNet Core API developed by Ruediger Willenberg [2] implements a subset of the GASNet Core API for x86 processors and FPGA heterogeneous systems. The GASNet Core API is a C library that provides an easy-to-reference PGAS view of the memory. It also provides a fast and efficient one-sided communication based on the AM paradigm. The THe_GASNet Core API extends the GASNet Core API support by creating a platform-specific network interface to integrate FPGA accelerators into software applications.

Figure 2.5 shows the system diagram of the THe_GASNet Core API layers with Willenberg’s contributions of highlighted in red outline. The software layers are similar to the original GASNet API shown in Figure 2.4, however, the network hardware has been extended to incorporate the hardware accelerators implemented on the FPGA. The network hardware provides the communication between the software and the hardware and is architecture specific, therefore the network hardware has to be updated whenever a new network is introduced to the system. This also means that the core API low-level functions that directly interact with the network also have to be updated. The functionality provided by the THe_GASNet Core API to software is provided to a hardware Accelerator Core by the Global Address Space Core (GASCore). This includes providing remote DMA for memory transfers. This allows the hardware nodes to use the same AMs to communicate with any other hardware or software nodes creating a uniform communication layer. The Programmable Active Message Sequencer (PAMS) is developed as a re-programmable control logic that allows the hardware nodes to initiate communication, and control the program flow.
Chapter 2. Background

Figure 2.6: MapleHoney Reconfigurable Computing Cluster (Source: R. Willenberg)

The THe_GASNet Core API has been tested on the MapleHoney cluster as shown in Figure 2.6. Since our work in this thesis is based on the THe_GASNet Core API, a more detailed background is provided here. The MapleHoney cluster consists of four Virtex 6-LX550T FPGAs that are located in the BEE4 Advanced Processing Platform [19], and four Intel Core i5-4440 quadcore CPUs. All the FPGAs in the BEE4 platform are connected in a ring topology. Each FPGA is directly paired with an x86 processor through a four-lane Generation 1 PCIe connection. Each x86 processor is fully-connected to each other through a 1Gbps network switch. This platform allows the FPGAs to communicate with each other directly through the ring topology, and allows a direct connection between the FPGA boards and the x86 processors through the PCIe controller.

Willenberg’s work provides a software library that implements a subset of the GASNet Core API functions called the THe_GASNet Core API, and hardware IP cores that provide the core API support to the FPGA. Similar to the GASNet Core API, communication between the parallel nodes is achieved using AMs. However, by extending the network interface to the hardware, the AMs are used to communicate with the hardware nodes. Software support for the AMs is achieved using a handler thread that services any incoming AM by running its appropriate handler function based on the handler ID. Each software node consists of a computation thread that is responsible for running the application and the handler thread. Besides the computation and the handler thread, the core API also forks one PCIe thread to listen to any incoming messages from the FPGA, and one TCP/IP thread to listen to any incoming messages from the network switch.

The core API AMs can be generalized into three types:

**Short messages** only carry a few integer arguments and no payload from the memory. They are
Figure 2.7: MapleHoney: internal design of a single FPGA (Source: R.Willenberg)

primarily used for signaling purposes.

**Medium messages**, in addition to the arguments, these messages carry a payload directly to a software or a hardware node. The destination node decides how the payload is processed. We use medium messages in the THe_GASNet Core API and THe_GASNet Extended API to provide a direct link between the software node and the hardware node.

**Long messages**, in addition to the arguments, these messages also carry a payload. However the payload is accompanied with the destination address to the remote node’s memory partition where the payload is transferred using a remote direct memory access (RDMA). Compared to the medium message, the destination node does not directly interact with the payload until after the payload is transferred to the memory.

These AMs are implemented using uniform packet formats that can be decoded by the software and the hardware nodes as shown in Appendix A.

Figure 2.7 shows the contents of a single FPGA chip. Each FPGA can host multiple nodes connected to each other through an on-chip network. A node on the FPGA is a parallel processing element that consists of an accelerator core, and the THe_GASNet Core API supporting IPs. The node can communicate to all the other nodes using the THe_GASNet Core API message passing functions. Node-to-node communication on a single chip is achieved using an on-chip network. The on-chip network is a fully-connected network that allows all the hardware nodes on the same chip to communicate with each other. The on-chip network also routes the messages to either PCIe, if the destination is on the x86 processor, or to an off-chip communication controller (OCCC), if the destination is on another FPGA. The OCCC connects all the FPGAs in a ring topology. In Figure 2.7 FPGA ring up and FPGA ring
down are the OCCCs that connect the FPGA to the other FPGAs. The on-chip network and the OCCC combined allow all the nodes running on all FPGAs to communicate with each other directly without needing x86 processor support. Additionally, each FPGA chip is connected to two 8GB DDR3 module that are used to implement the partitioned global address space for all the nodes in the FPGA chip.

The THe_GASNet Core API supporting IPs are the GASCore, and the PAMS. GASCore provides remote DMA support to the THe_GASNet Core API. All incoming messages to a node pass through the GASCore. If the message contains a payload, then the GASCore copies the payload to the destination address. If the hardware node wants to send an AM, then it requests the GASCore to read the memory and attach the payload to the outgoing message. PAMS is a re-programmable controller that provides communication, synchronization, and program flow control to the hardware node. PAMS contains a small instruction memory that is used to store low-level PAMS instructions shown in Appendix B. The instructions are created in the software application, and a software control node uses a medium message to program the instruction memory. PAMS contains a finite state machine that sequentially reads instructions from the instruction memory. These instructions allows the PAMS to initiate AMs, synchronize, and control the accelerator core. The instruction memory is re-programmable, so if the communication pattern changes for the application, the software control node can be used to re-program the instruction memory to account for the changes.

Unlike SHMEM+, the THe_GASNet Core API hardware IPs can respond to the core API calls and also initiate communication. This removes the hardware nodes from a purely slave role and reduces communication overhead of going through a x86 processors to reach other nodes. However, the THe_GASNet Core API faces similar challenges as the GASNet Core API because it only provides a minimal interface and does not support high-level functions. Any PGAS runtime compiler using other PGAS libraries cannot be ported easily without re-implementing all the functionality using the THe_GASNet Core API low-level functions.

Our work ports the THe_GASNet Core API to a Zynq SoC cluster and adds the ARM processor support as shown in Figure 2.8 highlighted in red outline. We updated the network hardware to account for the ARM-FPGA interface provided by the Zynq SoC, and modified the on-chip network to include the ARM processors. Furthermore, we implemented a high-level communication library called the THe_GASNet Extended API that is built using the core API functions to provide easy-to-program memory transfer functions. The PAMS block was updated to Extended PAMS (xPAMS) to enhance the functionality it provides for communication and control of the accelerator, and to add extended API support.

Ideally, we should be able to use the GASNet Extended API extended directly on top of the
The GASNet Core API, however, this cannot be done because of two reasons: a) the THe_GASNet Core API implements a subset of the original core API functions, b) the original GASNet Extended API does not fit into the memory available for the MicroBlaze. The high-level features developed for the THe_GASNet Extended API enhance the portability of the platform for other PGAS libraries by providing a bridge between the core library and the PGAS runtime libraries such as SHMEM, or UPC. The hardware and the software components of the THe_GASNet Extended API are described in detail in the following chapters.
Chapter 3

Hardware Implementation of the THe_GASNet Extended API

This chapter details the porting of the THe_GASNet Core API, introduced in the previous chapter, to an ARM-FPGA SoC cluster. The chapter also focuses on the modifications made on the core API to provide the extended API functionality. Section 3.1 describes the layout of the cluster and presents a high-level overview of the system implemented on the FPGA. Section 3.2 and Section 3.3 describe different tiers of networking implemented in the system to allow the parallel nodes to communicate. Section 3.5 presents the DMA block that allows communication between the ARM processors and the FPGA fabric. And finally, Section 3.6 describes the IP cores that provide support to the THe_GASNet Core API and the THe_GASNet Extended API on our platform.

3.1 System Overview

The testbed used to demonstrate the THe_GASNet Extended API contains eight ZedBoards [20]. Each ZedBoard contains a Xilinx Zynq-7000 All Programmable SoC [21] with 512MB of DDR RAM. The Xilinx Zynq SoC contains a dual-core ARM Cortex A9 Application Processor Unit and 28nm Xilinx programmable logic in a single chip. The data connection between the processing system (PS) and programmable logic (PL) is established through AMBA AXI interconnect. The interconnect contains four High-Performance (AXI_HP) ports reserved for communication with the DDR and can provide up to 1200MB/s for read and write each using 64 bits for the data. General Purpose (AXI_GP) ports provide AXI interface connections from the PS to the PL and from the PL to the PS at 600MB/s for
read and write each at 150MHz.

The ZedBoard is a development board that can boot Linux or Android and allows easy access to the PL for heterogeneous projects. For this research, we installed an ARM-compatible Linux called Linaro Ubuntu [22] on the PS. We configured the board using a partitioned SD card that contains the boot-image and the Linux filesystem. The boot-image contains the FSBL (first stage boot loader) that performs PS initialization, the configuration bitstream that programs the PL, and the U-boot Linux boot-loader that loads the OS when the board is turned on.

The ZedBoard platform is developed using Xilinx EDK 14.7 [23] because some of the IPs used from Ruediger Willenberg’s work [2] were developed under EDK. Converting to Vivado would be a logical next step so that the IPs can be used for future chip generations.

The bitstream generation takes the majority of the design cycle taking up to 40 minutes to generate the bitstream. Specific information about the PL cluster such as node IDs for all the parallel nodes are programmed from the software application, therefore, the same bitstream is loaded on all the ZedBoards. Currently the code for the hardware accelerator is handwritten, however, a High-Level Synthesis (HLS) tool can be used to generate performance critical code from a C source program.

The ZedBoard cluster has two tiers of networking: the off-chip network, and the on-chip network as shown in the Figure 3.1. The off-chip network refers to the board-to-board network that is achieved by connecting all the boards to a 1Gbps network switch. The on-chip network refers to the fully-connected 32-bit node-to-node network that is present in all the ZedBoards. The on-chip network implemented in
the ZedBoard cluster is similar to the one implemented in the MapleHoney cluster. However, unlike the MapleHoney cluster, the on-chip network on the ZedBoard does not connect directly to another off-chip ZedBoard.

PS-to-PL connection is achieved using the AXI-FSL DMA. The DMA is responsible for transferring a message from the DDR and to the on-chip network, or from the on-chip network to the DDR. The term “AXI-FSL” refers to the fact that it transfers the message from AXI bus (DDR) to the Fast Simplex Link (FSL) bus (on-chip network) or vice-versa. The on-chip network then routes the package to the correct hardware node.

A hardware node consists of a GASCore, a modified PAMS called Extended PAMS (xPAMS), and an accelerator core. The accelerator core is responsible for running the compute intensive task, whereas the GASCore and the xPAMS are the THe.GASNet Extended API supporting IPs. In the MapleHoney cluster, the GASCore is a remote DMA engine that is responsible for receiving and transmitting AMs. Since the message passing in the extended API is implemented using the core API messages, GASCore provides the same functionality to the THe.GASNet Extended API. Once a message reaches the destination node, the GASCore copies the payload from the message to the destination node’s partition in the memory through AXI\_HP ports. Note that in Figure 3.1 we use three AXI\_HP ports: one for the AXI-FSL DMA, and one for each hardware node. GASCore is also responsible for attaching the payload from the memory when the message is being transmitted by the hardware node. The xPAMS in the THe.GASNet Extended API is primarily responsible for initiating communication from the hardware, keeping track of incoming messages, synchronization, and controlling the accelerator core. To reduce the programming effort for the hardware nodes, xPAMS provides the support for the THe.GASNet Extended API functions, and provides a programmer-friendly way to program new handler functions. We have also added the support for automatically running the handler functions when a message arrives at xPAMS.

Since the ZedBoard only contains 512MB to be shared between the PS and the PL, we have limited the Linux OS to use only the first-half of the address space on the board (256MB), reserving the other half (256MB) for nodes located on the hardware. The second half of the memory is partitioned into equal sizes and distributed to the hardware nodes for their local address space as shown in Figure 3.2. For example, a project with two hardware nodes (node_0 and node_1) on the PL has the memory divided into 256MB for Linux, 128MB for node_0, and 128MB for node_1. This creates a bottleneck in the system where increasing the parallelism reduces the problem size each hardware node can run as shown in Figure 3.2(b) where each node has only 64MB. This bottleneck can be partially overcome using other Zynq-based boards that have separate DDR memories for the PL and the PS.
Chapter 3. Hardware Implementation of the THe_GASNet Extended API

(a) Two hardware nodes per board

(b) Four hardware nodes per board

Figure 3.2: DDR memory partition

Figure 3.3: ZedBoard Cluster
3.2 Off-Chip Network

ZedBoard has a 1Gbps Ethernet MAC with Direct Memory Access that is used to connect all the boards to a 1Gbps network switch as shown in Figure 3.3. The connection to the Ethernet port is established through the Linux OS running on the ARM processors. Having PS support for the Ethernet MAC allows us to easily implement a reliable TCP/IP network in software. This greatly reduces the risk of random packet drops, which is very difficult to detect and debug in a parallel infrastructure.

To use the TCP/IP network, we assign a static TCP/IP address to each board during the boot-up stage. The static TCP/IP address is used as the board identifier in the cluster. We also assign all software and hardware nodes in the system a unique node ID number. Nodes use the node ID to identify themselves and each other. The communication between the ZedBoards is achieved using stream sockets. Each board creates a server socket, and a client socket during THe_GASNet Core API initialization phase. A dedicated thread is used to read any incoming messages to the server socket, while the regular computation node uses the client socket to send an active message to another node.

<table>
<thead>
<tr>
<th>HOST</th>
<th>10.0.0.2</th>
<th>THREADS 4 7</th>
<th>BY_IP NO_LOOPBACK</th>
<th>FPGA 0 3</th>
<th>BY_IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>10.0.0.3</td>
<td>THREADS 12 16</td>
<td>BY_IP NO_LOOPBACK</td>
<td>FPGA 8 11</td>
<td>BY_IP</td>
</tr>
</tbody>
</table>

Listing 3.1: THe_GASNet Configuration File

A configuration file plays the key role in associating the TCP/IP addresses and the node IDs. The configuration file lists all the boards it has in the network by TCP/IP address, and lists all the associated node IDs to the IP address. An example of the configuration file shown in Listing 3.1. The configuration example lists two ZedBoard boards that have TCP/IP address 10.0.0.2 and 10.0.0.3. The board with TCP/IP address 10.0.0.2 has 8 nodes in total; nodes 0 to 3 are located in the PL region and nodes 4 to 7 are located in the PS (ARM processors) running as threads. The second ZedBoard with TCP/IP address 10.0.0.3 has nodes 8 to 11 in the PL, and nodes 12 to 16 in the PS. The same configuration file must be present on all the boards running the application and is loaded as an input when the application is run.

An incoming message to a board from the off-chip network is detected by a dedicated software thread that listens to the server socket. A thread reads the header of the AM first to determine the destination node ID and the length of the message. It then forwards the header as well as the payload to the appropriate node. In our example, a message from node 4 and to node 12 would go from the server socket of the board with TCP/IP address 10.0.0.2 to the network switch. The client socket on the board with TCP/IP address 10.0.0.3 would receive the message and pass it to software node 12.

One drawback of using the ZedBoard is that it does not allow the PL region to communicate using
the Ethernet MAC. This means that even in a hardware-only project, where all the nodes are located on the PL, the application is still run in the software to initialize the background threads that perform the message forwarding from the off-chip network to the on-chip network. If the board had an Ethernet MAC, or other communication port directly attached to the PL, then it would be possible to avoid using the software to do the communication. Again, the system used here is adequate to demonstrate the functionality of our system as the details of the communications is abstracted from the application level.

### 3.3 On-Chip Network

The on-chip network refers to the node-to-node communication from a PL node to another PL node, from a PL node to the PS, from a PS node to a PL node, where all the nodes are on the same device. On-chip communication is achieved using the NetIf network interface similar to THe\_GASNet Core API implementation on the BEE4 Platform. The NetIf block is responsible for routing all the packets in the network based on the node ID number and a routing table. As shown in Figure 3.4, the NetIfs on a chip are fully-connected. The channel between the NetIfs are FIFOs implemented using the Xilinx Fast Simplex Link (FSL). The FSL provides a unidirectional, point-to-point data streaming channel to the NetIf. Each NetIf has two FSL ports - one for incoming messages, and one for outgoing messages. A destination rank table is used to associate each channel at the output of the NetIf with a node ID range.
For example, if node IDs 0 to 3 are associated with Channel 0, then any message transmitted by the NetIf to any destination between 0 and 3 will be transferred through that channel.

Each AM is accompanied by a NetIf header that contains the source node ID (8 bits), the destination node ID (8 bits), and the size of the payload in words (16 bits). FSL buses also have a control bit that is asserted when the message contains the first word of the header. When the header reaches the NetIf block, the NetIf uses the destination rank table to forward the message to the appropriate channel. If the message is to be sent off-chip then we route that message to the NetIf associated with the ARM processors like the NetIf connected to the DMA in Figure 3.4. The NetIf outputs those data directly to a Direct Memory Access (AXI-FSL DMA) block, which transfers the message to the ARM cores to be routed off-chip.

### 3.4 Datapath Example

Using Listing 3.1 as an example, if a message is passed from Node 0 to Node 11, then the message is first put on to the on-chip network as shown in Figure 3.5. The arrow in the figure indicates the direction in which the data is traveling. The on-chip network forwards the message to the NetIf connected to the AXI-FSL DMA. The DMA block is used to transfer the message to the software. In the software application, the message is forwarded through the client socket of the board with TCP/IP address of...
Figure 3.6: DMA module to implement hardware-software communication

10.0.0.2 to the off-chip network. The client socket on the board with TCP/IP address 10.0.0.3 receives the message and passes it to the on-chip network in its device through the AXI-FSL DMA. The on-chip network will route the message to the appropriate hardware node. This message passing goes through three layers of networks and two DMA transfer processes, which adds significantly to the latency of the system. However, this problem can be addressed by switching to a different board where the PL can directly access the off-chip network.

Currently, we are using FSL buses because of their low-latency and low resource-usage. However, Xilinx has discontinued the usage of the FSL buses for its Vivado Product Line. Therefore, for any future research the infrastructure must be modified to use the AXI4 Stream instead of FSL buses. It means that the NetIf blocks will need to be redesigned to ignore the usage of the control bit for the interrupts.

3.5 DMA between PS and PL

The passing of active messages between the software nodes and the hardware nodes is accomplished through a custom Direct Memory Access unit called AXI-FSL DMA. As shown in Figure 3.6, the AXI-FSL DMA connects to the processing system through the AXI_GP port, and to the AXI_HP ports to
perform memory reads and writes of the DDR. The AXI-FSL DMA is custom-created because it has to be able to interact with the FSL channels connected to the on-chip network. The AXI-FSL DMA also contains three interrupts - one for read complete, one for write complete, and one interrupt to let the software know if a new message has arrived at the AXI-FSL DMA from the on-chip network.

AXI-FSL DMA is controlled by writing to its register space using the AXI4 Lite protocol through the AXI_GP port. AXI4 Lite does not support burst capabilities so the control commands have to be written one word at a time using separate AXI calls. Therefore, we’ve kept the register space to only four registers, two for read control and two for write control, to reduce latency to start a DMA operation. The following are the registers available in the AXI-FSL DMA register space:

**MEM2FSL_SA (Source Address Register)**: indicates the source address where the data is to be read from the memory

**MEM2FSL_CTRL_LEN (Control and Length Register)**: bits 15 to 0 indicate the number of words to be read from the memory, and bit 16 indicates if the control bit is to be asserted with the first word - used for indicating the packet header

**FSL2MEM_DA (Destination Address Register)**: indicates the destination address where the data is to be written in the memory

**FSL2MEM_LEN (Length Register)**: bits 15 to 0 indicate the number of words to be written to the memory

A dedicated thread in the ARM is created during the initialization phase that listens to the AXI-FSL DMA. POSIX read() and write() commands are used to control the DMA through a custom device driver. In the read phase, when a message arrives from the on-chip network to the DMA block, we raise an interrupt to wake up the thread sleeping on read(). The message contains the NetIf header and the AM. The interrupt handler function reads the NetIf header by writing to the FSL2MEM_DA register and the FSL2MEM_LEN register to initiate a transfer of one word (32 bits) to the memory. The header is decoded to determine the length of the AM. Then we write again to the FSL2MEM_DA register and the FSL2MEM_LEN register with an updated destination address and the length to transfer the rest of the message. The PS node to a PL node message transfer is initiated by first writing to the MEM2FSL_SA the source address where the message is located, and then writing the length of the message and the control bit to the MEM2FSL_CTRL_LEN register. Writing the length to the MEM2FSL_LEN register or to the FSL2MEM_LEN register starts the write or read operation respectively.
The destination address and the source address point to a DMA buffer created by the driver. To transfer a message from the software domain to the on-chip network, we first have to copy the data to the DMA buffer. Then, we provide the buffer address as the source address for the memory transfer. Similarly, when a message has to be read to a destination address, the destination address refers to the kernel buffer. So to do any read or write, we must copy the message to the buffer once using the `memcpy()` function, which reduces our bandwidth.

### 3.6 THe_GASNet Supporting IPs

The THe_GASNet Supporting IPs refers to the hardware blocks that help the compute units in performing the AM processing. The supporting IPs consist of the Global Address Space Core (GASCore), and the xPAMS.

Figure 3.7 shows the connection between the THe_GASNet Supporting IPs and the compute units for two cases: (a) if we are using the MicroBlaze soft processor [24] as the compute unit, (b) if we are using a custom accelerator as the compute unit. MicroBlaze is a soft-processor IP from Xilinx that provides support for the C programming language and a cross-compiler. While the MicroBlaze performs...
significantly slower than a custom accelerator or the ARM software threads, it can be used as a middle step to test the on-chip network connectivity or the GAScore. MicroBlaze provides flexibility and a relatively short design cycle to debug the hardware. When replacing the MicroBlaze with a custom accelerator, we want to maintain that flexibility in modifying the program flow, which is achieved using xPAMS as shown in the figure.

3.6.1 The Global Address Space Core (GASCore)

GASCore was developed as a remote DMA engine that manages message transfers to and from the PL nodes, either xPAMS or MicroBlaze, for THe_GASNet Core API. Since all the high-level communication functions use AMs, the THe_GASNet Extended API uses the same GASCore block that provides the AM support to the THe_GASNet Core API. Since we did not modify the GASCore, this section only provides a brief overview of its functionality. GASCore is connected directly to the on-chip network using the duplex FSL channel to send and receive messages. It connects to the memory through AXI_HP ports for memory transfers, and it connects to the xPAMS through another set of duplex FSL channels to send and receive AM requests.

**Receive Functionality**

AMs arriving at the GASCore are wrapped in the NetIf headers including an asserted control bit for the first word of the header. GASCore determines the AM type based on the header of the message. If the active message is of type Short or Medium, then the messages are passed as is to the hardware unit. If the arriving message is of type Long then the GASCore implements a memory transfer to the destination address found in the package header. Once the memory write is complete, the header is forwarded to the xPAMS or the MicroBlaze (depends on which hardware node is being used) along with any arguments attached to the message. The xPAMS or the MicroBlaze uses the header and the arguments to keep track of incoming messages for synchronization or signaling purposes.

**Transmit Functionality**

The xPAMS can initiate an outgoing message transfer by sending a request to the GASCore through the FSL channel. The request is translated by the GASCore to the packet format shown in Appendix A that can be read by either another GASCore or a GASNet software node. Short messages are simply passed through the GASCore since they do not require a memory transfer. While a medium message carries payloads, the payload is attached to the message by the xPAMS or the MicroBlaze and not read.
from the memory. Medium messages are also forwarded to the on-chip network. A Long message request prompts the GASCore to read from the memory and store the data in a large buffer capable of holding a complete long message. Once the memory read completes, the message is forwarded to the on-chip network.

3.6.2 Extended Programmable Active Message Sequencer (xPAMS)

If the Hello-Goodbye application presented in Section 2.1.3 needs to be changed to increase the number of threads, then the communication between the threads changes significantly. The “Hello” message now needs to be passed to more threads leading to a change in the for-loop limits. The barrier function must account for all the new threads as well. The for-loop to print the “Goodbye” must also be changed to account for the new number of threads. Due to the short compilation times for the software, these changes can be directly implemented in the source file and the whole application can be re-compiled. However, hardware design cycles do not give us the privilege to make such changes without going through a relatively longer synthesis, placement, and routing cycle leading to the loss of programming flexibility.

Changes in the number of nodes may not require a change in the main computation section of the application, but it requires changes in communication patterns depending on the new problem size, or the new number of nodes. So far the applications that have been implemented on the THe_GASNet Core API are not only compute heavy (justifying the use of accelerators), but also communication intensive. The THe_GASNet Core API addresses the loss of flexibility by creating a general finite state machine called the Programmable Active Message Sequencer (PAMS) that executes instructions from an instruction memory block called CodeRAM. The instructions can be placed in the CodeRAM through a medium message by a software control node. PAMS gives the hardware node the ability to communicate, synchronize, and control the accelerator IP core by the sequence of instructions that can be easily changed by changing the instruction memory.

PAMS only provides the ability to execute the instructions sequentially, which means that the handler function of an incoming message does not get executed when the message arrives. Instead, the handler function is only executed when the code in the PAMS reaches the handler section, which is essentially using polling to act on incoming messages. This can leave leaving the AM requests unserviced by PAMS for long periods of time. This also leads to the case that when programming the PAMS block, the programmer has to be aware of all the potential incoming messages, making PAMS hard to program when the number of incoming messages increase. The programmer has to handle both the send and receive functionality by carefully writing PAMS code. This is essentially two-sided communication. Extended
PAMS (xPAMS) is a updated version of PAMS that has the ability to respond to incoming AMs using interrupts to execute appropriate handler functions, thereby implementing one-sided communication. The xPAMS also adds THe,GASNet Extended API support by implementing a reply system for functions with handler IDs corresponding to standard extended functions such as `get()` or `put()`.

The xPAMS block interacts with the GASCore on one side, and the application core on the other side as shown in Figure 3.8. The connection to the GASCore is achieved using duplex FSL channels for incoming and outgoing requests. As mentioned in Section 3.6.1 GASCore is primarily responsible for interpreting the AMs and performing any DMA operations that may be required. The connection to the accelerator IP core is established through `set` and `wait for` channels. These custom channels provides us an interface to set the accelerator register space, and send start/stop signals to the accelerator core. It also allows the accelerator core to provide updates about its state, such as waiting/idle phase, computation phase, memory clear phase. While the custom channel has been working for us for various applications, future work remains to replace the custom channel with a standard protocol such as AXI4-Lite.

Figure 3.8 shows the major functional units in the xPAMS. The following describes the function of the these units:
• The **RX Decoder** is responsible for decoding incoming AM headers and providing information about the message such as handler function, type of the message, size of the payload, and the arguments it carries.

• The **Message counters** provide the ability to keep track of the number of messages arriving with a particular handler ID. In the THe,GASNet Extended API, message counters are used to block the program until a fixed number of messages with a specific handler ID arrive, thereby providing synchronization.

• The **Transfer counters** aggregate the number of bytes delivered by a particular handler. This allows us to keep track of very-large data transfers (over 8KB) that are transferred through multiple 8KB transfers.

• THe,GASNet Core API has a **Direct-Reply** functionality connecting the input of the PAMS to the output. The functionality has remained the same in the xPAMS and is used to implement `get()` functions for the THe,GASNet Extended API.

• **CodeRAM** is the instruction memory storage for the **Sequencer**. **Medium** messages are used to provide a link directly to the PAMS block, and a **medium message** with the code_load handler ID is used to write to the CodeRAM. The sequencer executes the instructions in the memory in-order, however, an incoming message can interrupt the sequencer to perform handler code.

• The **Reply Handler** was developed to implicitly handle any incoming requests that uses standard THe,GASNet Extended API functions.

• The **Synchronizer** provides synchronization between the messages by using counters that keep track of initiated AM requests, and completed AM requests.

**Handler Interrupt**

To support the interruptible handler, we have upgraded the CodeRAM to store handler functions separately. Originally, PAMS read the instructions in the CodeRAM sequentially, which meant that the handler functions had to be implemented within the program flow, making PAMS actively involved in the communication. For example, if Node 0 sends an active message that is used to start the accelerator on Node 1, then Node 1 needs to be aware of exactly when the data might arrive and has to be waiting for the message. Once the message arrives, it runs the code for starting the accelerator and then moves on to the next instruction. This leads to PAMS spending a significant amount of time in the wait state for the messages to arrive or messages are waiting in PAMS until PAMS reaches the handler instructions.
This not only forces the programmer to implement two sided communication, it also increases the code size with repetitive code if the message is to arrive multiple times in an iteration. Since we need to be aware of the incoming messages, all the PAMS blocks present in hardware nodes in the system need to be synchronized using a global barrier, thereby reducing the performance further.

To reduce the programming effort of the hardware node, we modified it to accept interrupts based on the handler ID. Having an interruptible sequencer allows us to execute the handler function as soon as the message arrives. This removes the need to make the sequencer wait for any incoming messages, or keep track of what it should be expecting. Any incoming message to xPAMS with a valid handler ID raises an interrupt in the sequencer. Note that interrupts are disabled unless the sequencer is loading a new instruction or is in a wait state. This removes the need to save the program state when executing handler functions. The sequencer uses the handler ID as an index to the CodeRAM where a pointer to the implementation of the handler is located. It then executes the handler function until it encounters the return instruction causing it to return to the main program and execute the next instruction. However, if the handler is just a signaling message where it does not need to execute a handler function, the pointer points to the return function, thus returning the sequencer to the main program in a single cycle.

Coming out of the reset state, xPAMS waits for the codeload handler function to finish, which is hard-coded into the CodeRAM at index 0 as shown in Figure 3.9. Indices from 1 to 256 are reserved for the handler function pointers, thereby supporting 256 handlers. The handler ID and the index in the CodeRAM have an offset of 1. For Handler ID 3, we look at CodeRAM index 4 for its function pointer. These function pointers contain the location of the handler function implementation in CodeRAM. Handler functions can be located anywhere from 256 to the end of the memory. The CodeRAM can be loaded using a medium message and the code_load handler ID. The message writes the handler pointers, application code, and the handler implementation. Once the CodeRAM is loaded with the application code, the xPAMS starts reading the instructions at index 257 and increments from there. Figure 3.9 shows an example of the instructions loaded into the CodeRAM.

In Figure 3.9 we have two valid handler IDs: 0 and 253. The handler pointers point to the implementation of handler functions for handler ID 0 and 253 at index 350 and 358 respectively. An incoming message with handler ID 0 will cause the sequencer to run the handler function starting from index 350 until it encounters a return instruction. It then resumes with the application code. Having a pointer at the handler ID index allows us to modify the application anytime without needing to reconfigure for the new position of the handler function. More than one handler ID can point to the same function, if required. The only requirement for the handler function is that it contains a return instruction at the end.
Figure 3.9: CodeRAM Memory Map
Chapter 4

THe_GASNet Extended API

This chapter describes the high-level communication library, the THe_GASNet Extended API, that we added for heterogeneous communication. Section 4.1 describes the programming model that is implemented. Section 4.2 provides a brief overview of the core API that was developed by Ruediger Willenberg, and the modifications made to allow the extended API support. Section 4.3 describes the extended API that provides the high-level communication for the heterogeneous nodes. Section 4.4 describes the PAMS Control Interface, which provides a similar set of high-level features to the hardware node. And finally, Section 4.5 describes an example of a point-to-point transfer to bring together different components described in this chapter.

4.1 The Programming Model

A programming model provides an efficient means to program applications while hiding the underlying hardware architecture and the memory layout from the programmer. Abstraction of low-level hardware-specific directives allows the programmer to focus on the core computation and the communication aspect rather than focusing on developing hardware-specific application code. With the increasing demand for heterogeneous platforms, a programming model should also follow a layered approach where different levels of codes are cleanly stratified into a well-defined hierarchy. Such an approach allows changes, upgrades, or extensions for a new architecture support without needing a change at the application level, thereby increasing programming model support for various heterogeneous platforms and portability of the model.

Generally a programming model can be implemented as a library, as an extension to an existing language, or an entirely new language. While an entirely new language focusing on a specific parallel
programming model and architecture gives the best performance, it presents a high learning curve and
needs to gain acceptance. It is also difficult to integrate other libraries that may have been developed
for a similar programming model into a new language. An extension to an existing language is the ideal
scenario but it relies on the language developers to add support for new architectures. Therefore, we
decided to focus on developing a library for our programming model. We decided to develop the library
in C as it can be easily integrated with other languages, and it provides greater control through low-level
function. Due to it’s extensive use, a C library would also ease any further development to the library
and provide it a greater exposure.

The work presented here extends the THe_GASNet Core API support to the ARM processor clus-
ter and to the ARM-FPGA SoC cluster. Through the THe_GASNet Core API the FPGA accelerators
located in the PL region can be easily integrated into a software application running on the ARM proces-
sors. Furthermore, we develop a programmer-friendly high-level library to program PGAS applications
called the THe_GASNet Extended API. Is is based on the THe_GASNet Core API. The THe_GASNet
Extended API is developed as an attempt to bridge the gap between the runtime library and the low-
level communication interface. It also serves as a bridging level between the THe_GASNet Core API
and other PGAS libraries such as openshmem, and SHMEM+ that allow the option to use the GASNet
Extended API as a conduit for implementing lower level functions.

4.2 THe_GASNet Core API Overview

This section presents the functions provided and the runtime structure implemented by the THe_GASNet
Core API, originally developed for the BEE4 Platform. While the core API functions have remained
the same when porting them to the ARM processors, we have modified the runtime structure to reflect
the ZedBoard cluster and support the extended API.

4.2.1 THe_GASNet Core API Communication Functions

As previously mentioned, THe_GASNet Core API supports three types of AM: short, medium, and long
messages. A short message carries one word of payload and the header is used for signaling purposes,
a medium message carries the payload directly to the destination node, and a long message carries the
payload to the destination node’s partition of the global address space. Table 4.1 shows the major
functions that are available in the THe_GASNet Core API.

- The utility function gasnet_init() is responsible for initializing the environment variables de-
<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Utility Functions</strong></td>
<td></td>
</tr>
<tr>
<td>Environment Init.</td>
<td><code>gasnet_init()</code></td>
</tr>
<tr>
<td>Resource Allocation</td>
<td><code>gasnet_attach()</code></td>
</tr>
<tr>
<td><strong>Active Message Request</strong></td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td><code>gasnet_AMRequestShortM()</code></td>
</tr>
<tr>
<td>Medium</td>
<td><code>gasnet_AMRequestMediumM()</code></td>
</tr>
<tr>
<td>Long</td>
<td><code>gasnet_AMRequestLongM()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_AMRequestLongAsyncM()</code></td>
</tr>
<tr>
<td>Long Strided</td>
<td><code>gasnet_AMRequestLongStridedM()</code></td>
</tr>
<tr>
<td><strong>Active Message Reply</strong></td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td><code>gasnet_AMReplyShortM()</code></td>
</tr>
<tr>
<td>Medium</td>
<td><code>gasnet_AMReplyMediumM()</code></td>
</tr>
<tr>
<td>Long</td>
<td><code>gasnet_AMReplyLongM()</code></td>
</tr>
</tbody>
</table>

Table 4.1: THe GASNet Core Functions

terminating the number of parallel nodes, and the current node’s ID.

- The `gasnet_attach()` function creates the handler threads, and the background threads that are needed to maintain on-chip and off-chip connections. It also allocates the memory region in DDR that acts as the software node’s partition in the PGAS programming model.

- The `gasnet_AMRequestShortM()`, `gasnet_AMRequestMediumM()`, and the `gasnet_AMRequestLongM()` perform an AM request to the destination node. These messages take as an argument to the function the handler function ID that is used to run the handler function at the destination node. However, the handler function has to be implemented by the programmer if using this library directly.

- The AM reply functions are initiated by a handler thread servicing the AM request.

- The difference between the `gasnet_AMRequestLongM()` and the `gasnet_AMRequestLongAsyncM()` is that the async long message requests that the handler function send a reply back. The data payload source memory is not safe to modify until a reply handler has been executed at the source node.

- The `gasnet_AMRequestLongStridedAsyncM()` function performs a long message transfer. However, the payload is not selected as a contiguous block. Instead, the payload consists of memory elements that are separated by a constant distance (stride) in the memory. For example, 0x0, 0x10, 0x20, 0x30 memory addresses have a stride of 0x10. The strided function provides a significant performance improvement over the traditional Long messages if the memory that is to be sent is non-contiguous. Strided functions can be implemented using a medium message and
a handler function in software. However, the hardware node uses PAMS/xPAMS to run handler functions and it does not have DMA functionality, therefore, a non-standard type AM was created.

- The reply message must only be initiated by the request handler and sends an AM to the requesting node.
- M at the end of the AM functions is to be replaced with the number of arguments sent with the message.

The AM functions shown above form a uniform message passing interface for all the nodes in the system including ARM software nodes, and the FPGA hardware nodes.

### 4.2.2 THe_GASNet Core API Runtime Structure

A major input to the THe_GASNet Core API application running in the ARM processors or x86 processors is the configuration file previously described in Section 3.3. The configuration file programs the software environments with the number of parallel nodes, and the number of boards running in the system. As seen in Listing 4.1 the configuration file associates the TCP/IP addresses for each device in the network with the node IDs running on the device.

<table>
<thead>
<tr>
<th>HOST</th>
<th>THREADS</th>
<th>BY_IP</th>
<th>NOLOOPBACK</th>
<th>FPGA</th>
<th>BY_IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.2</td>
<td>4 7</td>
<td></td>
<td></td>
<td>0 3</td>
<td></td>
</tr>
<tr>
<td>10.0.0.3</td>
<td>12 16</td>
<td></td>
<td></td>
<td>8 11</td>
<td></td>
</tr>
</tbody>
</table>

Listing 4.1: THe_GASNet Configuration File

In the case of a ZedBoard where the ARM processors and the FPGA fabric are on the same device, each unique TCP/IP address refers to one board. In Listing 4.1 the ARM processors are responsible for running the software nodes that have node IDs 4 to 7 and 12 to 16, and the FPGA fabric hosts the hardware nodes with node IDs 0 to 3 and 8 to 11. The nodes presented in the configuration file can be divided into the computation nodes, and a control node. Nodes 0 to 15 are computation nodes that are responsible for running the application. Node 16 is used as a control node responsible for programming the xPAMS, and synchronizing all the computation nodes.

If we go in-depth in a computation node, each software computation node contains two threads: a handler thread, and a main thread. The main thread is responsible for running the application code, while the handler thread is responsible for running handler code based on the handler ID of the incoming messages. In the THe_GASNet Core API, each handler thread has one FIFO channel where the incoming AM is placed. If a source node wants to send a message to a software node, then the source node must place the AM in the destination node’s FIFO channel. The handler thread of the destination node sleeps
until it detects a message in the FIFO (not empty). Once the AM is detected, the handler thread reads the handler ID of the AM and runs its appropriate handler function. In the THe.GASNet Extended API, the handler function of each request must send a reply AM. Only when the reply message comes from the destination node is the message transaction marked “complete” at the source node. However, if the FIFO channels of the source node are full, then the handler thread blocks until it is able to write the reply message to the source node’s FIFO. This leads to a potential deadlock when all the handler thread FIFOs for all the software nodes are full with AM requests, and the handler threads are blocked because they cannot send the reply message. To remove the deadlock, we implemented a two-channel solution where all the handler threads have two FIFO channels, a request-FIFO, and a reply-FIFO. Each incoming request is put on the request-FIFO, and each incoming reply is put on the reply-FIFO. This ensures that even if the request-FIFO is full with messages, we still have open FIFO channels to send reply messages, thereby servicing the requests.

In addition to the computation nodes and a control node, we have background threads that maintain the TCP/IP network and the PC to FPGA communication. For the BEE4 platform, the PC to FPGA connection is maintained through a PCIe thread. However, for the ZedBoard SoC cluster, we are using the AXI-FSL DMA for the PC to FPGA communication. Therefore, we have added the AXI-FSL DMA software interface to THe.GASNet Core API.

The TCP/IP thread listens to the Ethernet port for any incoming messages from the off-chip network, and the FPGA thread listens to the AXI-FSL DMA for any incoming messages from the on-chip network. These threads are created during the application run-time when THe.GASNet Core API reads the configuration file to determine the devices in the cluster. One issue with using THe.GASNet Core API on the ZedBoard is that it spawns the background threads from the application. In a case where we only want to use hardware nodes implemented on the FPGA fabric, we still need to run the software application on each board involved to maintain the TCP/IP network and the off-chip to on-chip communication.

4.3 THe.GASNet Extended API Overview

THe.GASNet Extended API is developed as a high-level communication library that provides higher flexibility in coding, and abstracts away the programming for the AM, and their respective handlers. Having a higher level communication library significantly reduces the coding effort, and provides expressiveness through the high-level message passing functions. The THe.GASNet Extended API also provides much more control over the status of the message. Since every message sent as a request is to
be followed by a reply function, we can keep track of every individual message sent. With very clearly
defined put and get memory transfer operations, it is easy to port applications written in other PGAS
libraries to the THe_GASNet Extended API software implementations as well.

While all the memory transfer operations are implemented using AMs, THe_GASNet Extended API
divides the memory transfers into three types based on the synchronization:

**Blocking Memory Transfer**: when the processing element initiates the memory transfer to a des-
tination node, and blocks (waits) till a reply message acknowledging the completion of transfer
arrives before proceeding with the next operation.

**Non-Blocking Memory Transfer (Explicit Handle)**: when the processing element returns after
initiating the transfer and proceeds with the next operation without waiting for the completion of
the operation. However, the call to these functions return a handle of type `gasnete_handle_t` to
represent the operation currently in progress. The handle holds the status of the transfer, and is
used to synchronize the operation. An explicit handle provides much finer control over the message
passing since for each message a new handle is generated.

**Non-Blocking Memory Transfer (Implicit Handle)**: when the processing element returns after
initiating the transfer similar to a non-blocking transfer with explicit handle. However, the call does
not return a handle. Instead, the user needs to use the synchronization function to synchronize
all the outstanding operations. These transfer operations are useful if we are transferring lots
of messages at once (on the order of megabytes) and we want to wait till all the transfers have
completed.

### 4.3.1 Handles

As mentioned in Section 4.3, non-blocking functions can be synchronized using explicit or implicit
handles. The GASNet Extended API implements the handle as an opaque type that can be implemented
based on the platform used. We use a high-level opaque function `gasnete_handle_t` to refer to all
the handles when the operations are in flight. Only the handler functions that modifies the operation’s
status know the type of non-blocking function used, explicit or implicit.

The explicit handle is implemented using a struct that contains two flags. The first flag contains the
type of the handle, implicit or explicit. The second flag contains the status of the message: `OP_FREE`,
`OP_INFLIGHT`, or `OP_COMPLETE`. The `OP_FREE` status means that the flag is not associated with any
message and is available in the pool of available handles. The `OP_INFLIGHT` status refers to the message
being in flight and not yet complete. A handle is marked \texttt{OP\_COMPLETE} when the reply message arrives carrying the handle address. The handle stays \texttt{OP\_COMPLETE} until the source node synchronizes using the handle. Once the source node uses the handle to check the status of the message, the status flag is changed to \texttt{OP\_FREE}.

Since each message sent using non-blocking transfer with explicit handle needs a new handle, we created a thread-specific array of explicit handles. According to the GASNet Extended API specification, the number of explicit handles provided should be wide enough to express \(2^{16} - 1\) different handle values. However, with limited resources on the ARM processor as well as on the MicroBlaze, we restricted the number to 256 handles for testing the functionality.

Each software node in the system has an implicit handle. The implicit handle consists of a struct with one flag and four counters. The flag denotes the type of the handle, implicit or explicit. The four counters are: \texttt{initiated\_get}, \texttt{initiated\_put}, \texttt{completed\_get}, and \texttt{completed\_put}. When a message is sent to a destination node, the sender node increments the respective \texttt{initiated\_*} counter atomically. Once a reply message arrives, the sender node increments the \texttt{completed\_*} counter. The status of types of requests in a software node is determined by the counter values. If the values in the \texttt{initiated\_*} counter and the \texttt{completed\_*} counter of the same type are equal then there are no pending requests. The xPAMS uses a similar structure in hardware to check if all the requests have been completed.

### 4.3.2 THe\textsubscript{e}\_GASNet Extended API Communication Functions

While the THe\textsubscript{e}\_GASNet Extended API is inspired by the GASNet Extended API \cite{gasnet}, we implement a subset of functions mentioned in the GASNet specification. The functions implemented in the THe\textsubscript{e}\_GASNet Extended API include the barrier functions, the memory transfer functions, and the synchronization functions. Implicit access region synchronization is a class of functions provided by the GASNet Extended API that allows grouping of more than one non-blocking implicit handle memory transfer to be associated with an explicit handle. These functions are not implemented in the THe\textsubscript{e}\_GASNet Extended API as the hardware currently cannot generate explicit handles. Additionally, we have also added support for strided memory transfers since the GASCore block inherited from the MapleHoney cluster had the support, and strided functions provided significant performance improvements when using non-contiguous data transfers. Although the current set of functions is sufficient for many operations, additional features can be added as the need arises. Table \ref{table:4.2} gives the list of all the functions provided by the THe\textsubscript{e}\_GASNet Extended API.
• The utility function `gasnet_extended_init()` is responsible for setting up the environment values for the extended functions.

• The `gasnet_attach()` function is a THe_GASNet Core API function that creates the handler threads, and the background threads that are needed to maintain on-chip and off-chip connection. It also allocates the memory region in DDR that acts as the software node’s partition in the PGAS programming model. While this function can be hidden under the THe_GASNet Extended API, it allows us to control the amount of memory that is allocated for the software node’s partition. Since the ZedBoard only provides 512 MB of memory, and only 256 MB for the ARM processors, this ensures that we do not run out of memory.

• Bulk get/put operations are the key memory transfer operations. Internally they call THe_GASNet Core API active message calls to implement the memory transfer. While source address, destination address, and the length of the transfer is provided by the application, the handler functions and the message types are generated based on the type of the call.

• The Synchronization Status functions allow us to check the status of the message transfers. They either return `GASNET_OK` if the operation is complete, or return `GASNET_ERR_NOT_READY` if the operation is not yet complete. When using explicit handles, the synchronization status functions tell the status of a particular handle. When using the implicit handle, the synchronization status function indicated whether all the messages of a certain type (get or put) are completed.

• The Synchronization Wait functions block the calling thread until the operations are complete. Similar to Synchronization Status functions, the Synchronization Wait functions block on the operation type (put or get) when using implicit handles, or they block on a specific handle when using explicit handle.

### 4.4 PAMS Control Interface

As mentioned in Chapter 3, the xPAMS is the hardware block that allows the hardware nodes to not only respond to the requests but also initiate AMs. The PAMS Control Interface refers to a set of high-level functions that can be used to program xPAMS to initiate a memory transfer. Note that the PAMS Control Interface functions are implemented using the PAMS instruction set that was developed for the THe_GASNet Core API presented in Appendix B.
## Chapter 4. The GASNet Extended API

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Utility Functions</strong></td>
<td></td>
</tr>
<tr>
<td>Environment Init.</td>
<td><code>gasnet_extended_init()</code></td>
</tr>
<tr>
<td>Resource Allocation</td>
<td><code>gasnet_attach()</code></td>
</tr>
<tr>
<td><strong>Blocking Memory Transfer</strong></td>
<td></td>
</tr>
<tr>
<td>Bulk put/get</td>
<td><code>gasnet_put()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_get()</code></td>
</tr>
<tr>
<td>Strided put/get</td>
<td><code>gasnet_puts()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_gets()</code></td>
</tr>
<tr>
<td><strong>Non-Blocking Memory Transfer</strong></td>
<td>(Explicit Handle)</td>
</tr>
<tr>
<td>Bulk put/get</td>
<td><code>gasnet_put_nb()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_get_nb()</code></td>
</tr>
<tr>
<td>Strided put/get</td>
<td><code>gasnet_puts_nb()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_gets_nb()</code></td>
</tr>
<tr>
<td>Synchronization Status</td>
<td><code>gasnet_try_syncnb()</code></td>
</tr>
<tr>
<td>Synchronization Wait</td>
<td><code>gasnet_wait_syncnb()</code></td>
</tr>
<tr>
<td><strong>Non-Blocking Synchronization</strong></td>
<td>(Implicit Handler)</td>
</tr>
<tr>
<td>Bulk put/get</td>
<td><code>gasnet_put_nbi()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_get_nbi()</code></td>
</tr>
<tr>
<td>Strided put/get</td>
<td><code>gasnet_puts_nbi()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_gets_nbi()</code></td>
</tr>
<tr>
<td>Synchronization Status</td>
<td><code>gasnet_try_syncnbi_puts()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_try_syncnbi_gets()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_try_syncnbi_all()</code></td>
</tr>
<tr>
<td>Synchronization Wait</td>
<td><code>gasnet_wait_syncnbi_puts()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_wait_syncnbi_gets()</code></td>
</tr>
<tr>
<td></td>
<td><code>gasnet_wait_syncnbi_all()</code></td>
</tr>
</tbody>
</table>

Table 4.2: The GASNet Extended Functions
46

Chapter 4. THE GASNet Extended API

### Table 4.3: PAMS Control Interface

<table>
<thead>
<tr>
<th>Category</th>
<th>Example Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Blocking Synchronization (Implicit Handler)</td>
<td></td>
</tr>
<tr>
<td>Bulk put/get</td>
<td>pams_put_nbi()</td>
</tr>
<tr>
<td></td>
<td>pams_get_nbi()</td>
</tr>
<tr>
<td>Strided put/get</td>
<td>pams_puts_nbi()</td>
</tr>
<tr>
<td></td>
<td>pams_gets_nbi()</td>
</tr>
<tr>
<td>Synchronization Wait</td>
<td>pams_wait_syncnbi_puts()</td>
</tr>
<tr>
<td></td>
<td>pams_wait_syncnbi_gets()</td>
</tr>
<tr>
<td></td>
<td>pams_wait_syncnbi_all()</td>
</tr>
</tbody>
</table>

The PAMS Control Interface is used by the control node in the software to create the xPAMS program code. The xPAMS instructions are stored in an array, which is then transferred to the xPAMS through a medium message. The ability to control the hardware flow using medium messages from software greatly increases the ease to modify or re-program the cluster without having to go through the hardware design cycle.

The functions currently provided for the xPAMS can only initiate Non-Blocking Memory Transfer with implicit handles. The synchronization functions rely on counters to ensure that all the requests have been fulfilled. The support for explicit handles can be added for the xPAMS using a `gasnete.handle_t` array represented in the on-chip memory. However, the circuity for maintaining and adequately using the library requires another upgrade of xPAMS that remains part of the future work. The current set of functions provides a minimal interface to study the impact of programming the xPAMS with high-level functions.

The goal of the PAMS Control Interface is to provide a similar set of high-level functions as the software library. Eventually, this correlation can be exploited by an HLS tool or a PGAS runtime compiler to generate xPAMS instructions based on the software application.

### 4.5 Point-to-Point Transfer Operation

This section gives a brief overview of a full memory transfer operation. Figure 4.1 (a) shows an example of a `put()` operation, and Figure 4.1 (b) shows an example of the `get()` function. The only difference between the `put()` and `get()` operations is that in `put`, we send a payload first and then accept a short AM reply, whereas in `get()` we sent a short AM request first and then accept a long AM reply with the payload.

The following list goes through the memory-transfer operation in detail. The main stages are numbered as 1, 2, 3, etc. and the sub-items (a), (b), (c), etc. describe options that can be executed depending
Figure 4.1: Point-to-Point Memory Transfer Example
on the type of transfer and the destination node.

1. NodeA calls the `gasnet_put_nb()` to put the message to the remote partition, or `gasnet_get_nb()` to fetch the data from a remote partition:

   (a) If the message is to be passed using explicit handles, then the main thread first traverses the handle pool array to find a handle with status `OP_FREE` and marks it `OP_INFLIGHT`.

   (b) If the message is to be passed using implicit handles, then the main thread will increment the appropriate `initialize_*` counter.

2. The address of the explicit or implicit handle is then attached to the AM as an argument, along side any payload that has to be carried:

   (a) If the message is sent with a put operation, a `long` AM is sent with a payload along with the destination address where the memory is to be copied.

   (b) If the message is sent with a get operation, a `short` AM is sent with the remote get function attached as arguments.

3. The message is then ready to be sent to the appropriate node based on the destination address:

   (a) If NodeB is a local software node, then the message is copied to the destination handler’s FIFO.

   (b) If NodeB is with the local FPGA, then the message is passed to the AXI-FSL DMA to be forwarded to the hardware node.

   (c) If NodeB is remote, then the message is passed to the off-chip network through TCP/IP.

4. At the NodeB, the handler thread will perform the handler functions based on the handler ID found in the AM. If the message sent was a put message, we send a `short` reply, and if the message sent was a get message we send a `long` reply with the requested data as payload:

   (a) If the message sent is a put operation, then the handler function sends a reply through a `short` AM and attaches the last three arguments of the incoming message (handle’s address) to the `short` reply AM as arguments.

   (b) If the message sent is a get operation, then the handler function executes the arguments of the request AM, which will run the get operation on Node B node. The handler function then sends a `long` reply AM with the payload and the (handle’s address) as arguments to Node A.
5. NodeA then reads the arguments on the reply function as an address, which points to the appropriate handle, and marks it `OP_COMPLETE` or increments the `completed_*` depending of the handle type.

A successful synchronization of the get functions means that the results requested are now ready to be read from the destination address. Note that the source memory may or may not be updated when the message was requested. The use of global barriers is recommended to ensure that the data request sends the updated values. Likewise, a successful synchronization of the put function means that the data has been sent to the destination and it can now be modified at the source address.
Chapter 5

Results and Analysis

The goal of this chapter is to characterize and evaluate the architecture using a series of tests and an application to identify the areas of improvement for the future work. Section 5.1 presents the result of latency and throughput tests of various blocks in the system, such as the on-chip network, the off-chip network, and the AXI-FSL DMA, in an attempt to quantify the bottlenecks. Section 5.2 presents the result of the Jacobi method for Heat Transfer Equation implementation using the core API as well as the extended API. This section provides the overall performance of the cluster and identifies the overhead added by the extended API. It also presents a code comparison between the core API and the extended API highlighting different approaches provided by both libraries.

5.1 Microbenchmarks

The performance of a system is directly affected by the performance of its individual components. By studying the latency and the throughput of the performance critical blocks, we can identify the bottlenecks in the system. These bottlenecks help explain the overall performance of the system and provide areas of improvement for future work.

5.1.1 Latency

The goal of the first test is to study the latency and the bandwidth of the message-passing architecture implemented in the ZedBoard. This is done by observing the latency of a ping-pong short message. In the ping-pong message, a source node sends a short message to the destination node. Upon receiving the short message, the destination node sends a short reply back to the source node. A short message from the THe,GASNet Core API is used to get the latency of the message-passing functions only and not include
any added latency of managing the handles introduced by the THe_GASNet Extended API functions. The source node is implemented on the ARM processors while the location of the destination node varies to test the latency of: the off-chip network, the `memcpy()` function, and the PL architecture. Since the tests are implemented on the ARM processors, the software layer can add to the latency depending on the background tasks running on the OS. Therefore, the test was repeated 1000 times to get the average latency. The `time.h` library was used to measure the elapsed time (`CLOCK_MONOTONIC`) on the ARM processors. Table 5.1 shows the results of the two-way short ping-pong.

<table>
<thead>
<tr>
<th>Two-way short ping-pong</th>
<th>Communication Type</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software node-software node</td>
<td>Local</td>
<td>42.8</td>
</tr>
<tr>
<td>Software node-software node</td>
<td>Remote</td>
<td>119.6</td>
</tr>
<tr>
<td>Software node-hardware node</td>
<td>Local</td>
<td>176.70</td>
</tr>
<tr>
<td>Software node-hardware node</td>
<td>Remote</td>
<td>363.20</td>
</tr>
<tr>
<td>Hardware node-hardware node</td>
<td>Local</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Table 5.1: Latency of on-chip communication and off-chip communication

- The Software node-software node local communication refers to the message passing where both the source and the destination nodes are located on the same device (local). A `memcpy()` function is used to transfer the data from the source node’s address space to the request-FIFO of the destination node and the other way around as well. Since the data does not leave the device, the latency is very low.

- The Software node-software node remote communication refers to the message passing where both the source node and the destination node are implemented in software, but are located on different devices (remote). Therefore, the message travels through the off-chip network implemented using a 1 Gbps network switch. The experiment was not performed in a controlled environment, so other devices using the network switch may affect the latency of the off-chip network. Since the messages are going off-chip, it takes three times the local `memcpy()` transfer.

- The Software node-hardware node local communication refers to the message passing where the source node is located on the ARM processor, and the destination node is located in the PL of the same device. The latency shown in Table 5.1 for this transfer includes the latency of the DMA transfer, the on-chip network, and the GASCore block. The high latency of this message passing can be attributed to the multiple `memcpy()` calls required to transfer the data to and from the DMA buffer before being sent to the on-chip network by the DMA block. A message going to the on-chip network from the software layer needs to be transferred to the DMA buffer using the `memcpy()` function, and a message coming from the on-chip network to the software layer has
to be transferred from the DMA buffer to the buffer read by the application twice, once for the header and once for the data. Beside the `memcpy()` function, a message passing between the software node and the hardware node also spends time transferring the control signals between the user space and the kernel space one word at a time for the AXI-FSL DMA.

- The Software node-hardware node remote communication refers to the message passing where the source node is located on the ARM processor, and the destination node is located in the PL of a different device. In this communication we use the off-chip network, the on-chip network, and the AXI-FSL DMA. Ideally, the time taken for this ping-pong should be the sum of time taken by the software node-software node remote communication, and the software node-hardware node local communication, which is about 300$\mu$s. However, the time that was recorded for the ping-pong was 363.20$\mu$s. When using only the hardware node on the board, we still run the software application that forks a FPGA thread and a TCP/IP thread to poll for the messages from the off-chip network and the on-chip network. However, the main application thread that forks the background threads then blocks in a while loop, which adds time to the context switching. Additionally, when running the software node-software node remote communication test, we did not fork the FPGA threads leading to better performance due to reduced context switching. These factors combined, along with uncontrolled network switch adds to the additional time spent.

- The hardware node-hardware node local communication refers to the message passing where both the source and the destination node of the hardware are located on the same device. The test measures the time it takes for the response message to arrive after the request message has left the PAMS block of the source node. The results were obtained using Chipscope Logic Analyzer, which provides the values in the FSL channels for each clock cycle. A full ping-pong between two local hardware nodes takes 39 cycles at 100MHz. The majority of the time is spent in the source and the destination GASCore, which decodes each incoming message either from the NetIf and PAMS to determine the type of the transfer and any arguments attached before forwarding it to the recipient.

The Hardware node-hardware node remote communication time cannot be measured at the moment because the Chipscope Logic Analyzer does not have enough capacity to store all the samples for the ping-pong. However, a new micro-benchmark that uses a counter that starts counting when we send the first message, and stops counting once it gets a reply back can be used to accurately measure the communication time for the future work. Based on previous experiments, the time taken for the hardware node to the hardware node communication can be estimated to be the sum of software node-hardware node...
Chapter 5. Results and Analysis

5.1.2 Throughput of the AXI-FSL DMA

Figure 5.1 shows the throughput of the AXI-FSL DMA measured over various data-sizes. The figure shows the DMA Read operation and the DMA Write operation. When the AXI-FSL DMA transfers the data from the memory to the on-chip network, the operation is defined as a DMA Read. When the AXI-FSL DMA transfers the data from the on-chip network to the memory, the operation is defined as a DMA Write.

To measure the throughput of the DMA Read operation, we transfer a long message from a software node to a hardware node. We then measure the time it takes for the AXI-FSL DMA block to transfer the message from the memory to the on-chip network. The AXI-FSL DMA sends an interrupt to the software thread once the transfer operation is complete. The test is repeated with various data sizes to collect the throughput of the DMA Read operation over a range of data sizes.

To measure the throughput of the DMA Write operation, we program the PAMS to transfer a long message to a software node. We then measure the time it takes for the AXI-FSL DMA from the on-chip network, the AXI-FSL DMA sends an interrupt to the software control thread (FPGA thread). The FPGA thread then orders the AXI-FSL DMA to transfer the header of the message. Based on the size of the payload determined from the header, the FPGA thread re-programs the AXI-FSL DMA to transfer the payload node remote communication, and software node-hardware node local communication, which is about 539.90µs.
from the on-chip network to the memory. We measure the time it takes for the AXI-FSL DMA to transfers the data from the on-chip network to the memory to provide us the throughput of the DMA Write operation. The test is also repeated with various data sizes to collect the throughput results over a range of data sizes.

The size of the data ranges from 32 bytes to 4 megabytes (MB). The upper limit of the data size was capped at 4 MB because the throughput of the DMA Read and the DMA Write stabilizes by then. As shown in Figure 5.1, the AXI-FSL DMA has the peak performance of 165 MB/s for DMA Write when transferring 8 KB of data, and 123 MB/s for DMA Read when transferring 256 KB of data. Results for the latency for each transfer is shown in Appendix C.

For a message transfer from the on-chip network to the memory, the FPGA thread first reads the header, it then reads the payload (determined by the header). For less than 8 KB message transfers, we only measure the time taken to transfer the payload since it is a more accurate representative of the AXI-FSL DMA throughput. However, for greater than 8 KB message transfers, we count the time it takes to transfer of the first payload, the time it takes for the FPGA thread to get the interrupt from the AXI-FSL DMA for another packet, the time it takes to transfer the second header, and the time it takes to transfer the second payload, and so on. This significantly reduces the throughput number obtained for the DMA Write operation, however, this is not due to a slowdown in the AXI-FSL DMA, but it is because of how the time is collected. However, currently this is the only method to measure the DMA Write time since we cannot distinguish between independent long messages, and long messages that are transferring data greater than 8 KB. The throughput stabilizes around 130 MB/s for DMA Write for data transfers greater than 16 KB.

For a DDR to the on-chip network transfer, we measure the time it takes from sending the control signals of the AXI-FSL to the return of an interrupt from the AXI-FSL DMA marking the completion of the operation. For message transfers greater than 8 KB, we only measure the time it takes to transfer the 8 KB message multiple times, which allows us to accurately determine the throughput for the read operation from the DMA showing no significant change in the DMA throughput. The throughput stabilizes around 113 MB/s for DMA Read for data transfers greater than 16 KB.

### 5.1.3 Throughput of the Datapath

Figure 5.2 shows the result of the throughput of the data-path when transferring messages between a software node and the hardware node. The full data-path includes the AXI-FSL DMA, the on-chip network, and the GASCore.
Figure 5.2: Throughput of the software-to-hardware datapath

To measure the throughput of the software node to the hardware node data path, we program the PAMS block to send a short message to the software node when its internal timer (counter) reaches a specific value, however, the timer keeps counting. This process helps in synchronizing the software node and the hardware node. The software node then sends data through long messages. Once the message arrives at the PAMS block, PAMS sends another short message with the timer value to the software node. Using the initial timer value from the first short message, and the new timer value we can determine the time it took for the data to reach the PAMS block.

To measure the throughput of the hardware node to the software node data path, we program the PAMS block to wait until its internal timer reaches a specific value that is also known to the software node. The PAMS then sends the data through long messages to the software node. Once all the messages arrive at the software node, the software node sends a short request to the PAMS to provide the software node with the current timer value. This would mark the data transfer complete. Using the start timer value, and the new timer value sent after the data transfer is over, the time taken to transfer the message is determined.

A large message traversing the full data-path between the software node and the hardware node will perform two memory operations: one performed by the AXI-FSL DMA, and one performed by the GASCore, which reduces the throughput of the data-path greatly. The Software node to the hardware node throughput peaks at 59.1 MB/s when transferring 16 MB of data, and the hardware node to the
software node throughput peaks at 39.4 MB/s for messages between 256 KB to 16 MB. The reason for the low throughput of the software node to hardware node transfers can be attributed to the overhead of the software libraries and the OS layer when initiating a data transfer from the software node. Whereas, the data transfer initiated by the PAMS occurs within a few clock cycles. The dip in the throughput at 64 MB can be attributed to the buffering in the software layers, since the performance of the on-chip network and THe_GASNet supporting IPs are deterministic. However, 64 MB is the largest size that can be allocated by a software thread on the ARM processors, so further testing to determine the throughput of data sizes greater than 64 MB is not possible.

5.2 Heat Transfer Equation Case Study

To measure the performance of the system for a meaningful application, we have implemented the Jacobi iterative method for solving the heat transfer equation iteratively. The Heat Transfer Equation, shown in Equation 5.1 is used to calculate the rate of change in temperature of a given region over time.

\[
\frac{du}{dt} = \alpha \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right) \tag{5.1}
\]

where, \( \frac{du}{dt} \) is the rate of change temperature over time for a given location \((x, y)\) on a 2D plane and \(\alpha\) is thermal diffusivity constant which measures how quickly the material reacts to the change in temperature. For our application, we have set the value of the \(\alpha\) to 1.

\[
u_{t+1,x,y} = \frac{u_{t,x,(y-1)} + u_{t,(x-1),y} + u_{t,(x+1),y} + u_{t,x,(y+1)}}{4} \tag{5.2}
\]

\[
\sqrt{\sum_{x,y} (u_{t+1,x,y} - u_{t,x,y})^2} < \epsilon \tag{5.3}
\]

The heat transfer equation can be solved using an iterative process as shown in Equation 5.2. The rate of change in heat at a location \((x, y)\) at time \(t+1\) is based on the temperature of its surrounding locations \((x, y-1), (x-1, y), (x+1, y), (x, y+1)\) at time \(t\). The convergence condition of this equation is computed as the square root of the sum of the square of the differences between the old values and the new values as shown in Equation 5.3.

Parallel implementation of the Jacobi method divides the 2D test surface into sections that can be distributed among the parallel nodes. Figure 5.3 provides an example of a 12x12 surface that is divided among nine nodes with each node computing the heat transfer for 16 cells. The parallel nodes can carry
Chapter 5. Results and Analysis

Figure 5.3: 2D Surface Partitioning (Source: R. Willenberg)

Figure 5.4: Node-to-Node Communication between Iterations (Source: R. Willenberg)
out the computation independently until they hit the boundary regions of their respective sections. The parallel nodes need get the outer rings from the four surrounding neighbors, shown in Figure 5.4, to calculate the rate of change in temperature at their respective boundaries. While the 2D surface can be parallelized, the parallel blocks need to communicate to accurately compute the heat transfer equation.

Each divided subsection of the surface is stored in the local partition of a parallel node. Communication functions are used to transfer the edges from a local partition to a neighboring node’s partition using THe_GASNet Extended API functions such as put() or get(). All the parallel node memories are organized the same way, so using relative addressing we can locate the edges of a remote node’s surface. In the case of the put() or the get() functions, we use relative addressing to communicate the edges between the parallel nodes. Initially, all the nodes initialize their memories to zero, and a control node then places static heat values at certain locations in the grid referred as poles. This means that while the heat is spreading to the system iteratively, constant heat is being applied through these poles. Note that the pole values, as well as the heat values at each cell, are represented using a 32-bit fixed-point number.

This application was chosen primarily because it was used to test the performance of the THe_GASNet Core API on the BEE4 platform. Continuing to use the Jacobi method for the THe_GASNet Extended API allows us to test the performance of the extended functions vis-a-vis the core functions. It also allows us to compare the performance differences between the BEE4 platform and the ZedBoard cluster.

5.2.1 Jacobi Software Implementation

Algorithm 1: Jacobi iterative method software implementation

```plaintext
1: procedure SPMD_MAIN
2: Compute partition size
3: Allocate blocks EVEN, ODD
4: Initialize blocks to 0
5: while Iteration < 1024 do ▶ iteration loop
6:   Send edge cells from EVEN to neighbor nodes ▶ communication
7:   Wait for all the edge data to arrive ▶ barrier
8:   for all local cells in block ODD do ▶ computation
9:     Read neighboring cells from block EVEN
10:    Compute local heat based on neighboring values
11:   Write output to ODD
12:   Wait for all nodes to finish computation ▶ barrier
13:  Swap ODD and EVEN
14: Return
```

Algorithm 1 presents the pseudocode of the Jacobi method for heat transfer over a two-dimensional
surface implemented for the x86 processor and the ARM processors. All parallel nodes in the system work on a section of the test surface, which is represented by a 2D matrix. Each node allocates enough memory to hold two copies of their section of the test surface, ODD and EVEN, as shown in line 3. This allows us to hold the temperature values for time $t$ and time $t+1$, which is used to calculate the convergence value based on Equation 5.3. When doing the computation, we read from the EVEN copy of the memory and write the output to the ODD copy. When we are done updating the ODD copy of the test surface then we swap the pointers to the EVEN and ODD, so the ODD copy has the old version again and the EVEN copy has the newer version. The first barriers at line 7 ensure that all the memory has been updated with the new edge values, and we can proceed with the computation. The second barrier ensures that the computation phase in all of the of nodes has completed, and it is safe to fetch or put data in the destination node’s memory. In our experiments we test the performance of the software and the hardware implementation over multiple grid lengths. The length of the grid affects the time it takes to converge; a smaller grid will converge faster. Therefore, we run the application for 1024 iterations to compare the performance of the hardware and the software nodes over varying grid sizes for a fixed number of iterations.

5.2.2 Jacobi Hardware Implementation

Algorithm 2: Jacobi iterative method - control node

1: procedure SPMD\_MAIN
2: Compute partition size
3: Program all xPAMS blocks
4: while Iteration < 1024 do ▷ iteration loop
5: Wait for communication phase to end ▷ barrier
6: Accumulate sum-of-squares from all nodes ▷ barrier
7: if Iteration = 1023 then
8: Signal nodes to stop
9: else
10: Signal nodes to continue
11: Return

The software algorithm has two key phases: the computation phase, and the communication phase. When transferring the application to the hardware, we provide both of these functionalities to the hardware. The computation code for the application is implemented through a custom IP core. The IP core currently used in our system is hand-written, however, ideally these cores can be generated through high level synthesis tools such as Vivado HLS [26]. The communication is provided by the xPAMS block. A clean partition between the communication and computation tasks among cores ensures that
Algorithm 3: Jacobi iterative method - computation nodes

1: procedure xPAMS\_MAIN
2: \hspace{1em} Wait for the code\_load signal
3: \hspace{1em} Wait for memory to be initialized to 0
4: \hspace{1em} while continue do \hfill \triangleright iteration loop
5: \hspace{2em} Send edge cells from EVEN to neighbor nodes \hfill \triangleright communication
6: \hspace{2em} Wait for all the edge data to arrive \hfill \triangleright barrier
7: \hspace{2em} Signal the hardware core to commence computation \hfill \triangleright computation
8: \hspace{2em} Wait for the computation phase to finish
9: \hspace{2em} Read the convergence value from the hardware core
10: \hspace{2em} Send the convergence value to the control node
11: \hspace{2em} Wait for the continue signal from control node \hfill \triangleright barrier

any change in the program flow can be implemented in the hardware by re-programming xPAMS through an AM.

A control node is run in the ARM Processor to implement the xPAMS programming functionality, and to control the program flow. Algorithm 2 shows the pseudocode of the control node. The control node calculates the size of the grid that will be run on each of the parallel nodes. It uses the size of the grid, and the number of parallel nodes to generate xPAMS machine code. Once the xPAMS is programmed with its instructions, the control node is mainly responsible for keeping track of the program through global barriers. The accumulation of the sum-of-squares from each node, and then sending continue signal to each node acts as a barrier as well. The control node is also used to re-program xPAMS if there are any changes in the communication patterns to account for the changes.

The xPAMS code shown in Algorithm 3 goes through the majority of the xPAMS functionality. Coming out of the reset state xPAMS waits for the code\_load signal. We program the xPAMS to wait for a memory clear signal from the hardware core. The hardware core is designed to initialize all the memory in its partition to 0. In the communication phase, xPAMS is used to generate AM requests to other nodes to either put the data in their memory or fetch the data from them. As the communication phase ends, we wait at the barrier to ensure that our memory has been updated and is safe to use. The computation phase is implemented by the hardware core and the xPAMS waits for a done signal from hardware core through the custom channel. Once the done signal arrives, the xPAMS reads the contents of the output register from the hardware core, and passes it to the control node. Based on the control node reply, we either continue or just wait at the current state.

The computation code is implemented by a hardware IP core developed by Ruediger Willenberg. The hardware core directly connects to the memory through AXI\_HP ports. The hardware core reads three rows of the grid from the memory and stores them in a local buffer. It then computes the new values for
the second row sequentially, which is immediately written back to the memory. It also accumulates the difference between old values and the new values in order to determine the convergence point. While the hardware block writes back the second row values, it reads the fourth row and stores it into the local buffer. Once the block has written the updated values of the second row back to the memory, it reruns the computation with the third row. The process repeats until all the rows in the memory are read. The hardware core also partitions the local memory into ODD and EVEN regions, similar to the software application. It reads from the ODD region and writes to the EVEN region; after each iteration it swaps the pointers to the ODD and EVEN regions. However, the partitions are used to only store the 2D surface values, the convergence values are stored in local registers that are accessed by the xPAMS block.

5.2.3 Code Comparison

This section provides a comparison between the Jacobi method implementation using the THe_GASNet Core API and the THe_GASNet Extended API for a software node to show the differences between the core API and the extended API.

Listing 5.1 shows excerpts from the Jacobi method implemented using the core API. Lines 1 to 19 initialize the handler functions that will be used to handle incoming AMs from other software nodes and puts them in a handler table. The handler table is used to associate the handler ID with the handler function. In the table, the handler IDs are the hc_L0_update_aura and the hc_S4_send_stats. These handler IDs are associated with the L0_update_aura and the S4_send_stats handler functions respectively. The handler table is shortened significantly to keep the example size smaller. Line 22 to line 27 send the long message to the neighboring nodes. The outgoing message must contain a handler ID that is used to run the appropriate handler function at the destination node. Line 29 sends the convergence values back to a control node that determines if the other nodes need to iterate. The implementation of the handler functions are shown in line 33 to 49.

In the core API AMs, the handler functions need to be created by the programmer for all the applications. While this gives the programmer greater control over the AMs, but it makes programming harder and repetitive as the same functionality may need to be implemented for other applications.
Chapter 5. Results and Analysis

Listing 5.1: Jacobi Method message passing using the TH{e}GASNet Core API
Listing 5.2: Jacobi Method message passing using the THe_GASNet Extended API

Listing 5.2 shows excerpts from the Jacobi method implemented using the extended API. The key difference compared to the core API is that the programmer can use the standard set of functions for message passing as shown in lines 17 to 20 where we use standard put() functions to transfer the data.

To keep the implementation similar to the xPAMS source code, we use the core API functions for the passing convergence values as well as shown in line 23. As you may recall from Section 5.2.2, the Jacobi hardware IP core stores the convergence results in local registers, which can only be accessed by the xPAMS. Since the convergence values are not stored in the PGAS partition of the hardware node, they cannot be transferred using THe_GASNet Extended API functions.

Listing 5.3 shows the ideal implementation of the Jacobi method implemented using the extended API. Compared to Listing 5.2, the ideal version stores the convergence values on the PGAS partition as shown in lines 24 and 25. The results are then transferred to the control node using the extended functions as shown in line 26 and 27. The code provides a much easier method for programming the
Listing 5.3: Jacobi Method message passing using the THe_GASNet Extended API ideally parallel nodes to the programmer by providing a standard set of high-level functions for message passing.

5.2.4 Experimental Setup

As mentioned previously, the aim of the Jacobi method is to benchmark not only the THe_GASNet Core API and THe_GASNet Extended API, but also the Zynq SoC performance for applications using the PGAS programming model. The Jacobi method for solving the heat transfer equation was run on three systems: the ARM cluster, an x86 processor, and the ARM-FPGA SoC cluster. Both the ARM cluster and the ARM-FPGA cluster were implemented using ZedBoards that contains Zynq XC7Z020 SoC [20] and 512MB of DDR3 memory. The ARM cluster only uses the ARM processors on the ZedBoard, while the ARM-FPGA Cluster uses both the PS and PL region to implement a heterogeneous cluster. The ARM cluster and the x86 processor are used to test the performance of the software library and provide a comparison between the core API and the extended API. Using the x86 processor also allows us to compare the result of the ARM cluster and the ARM-FPGA SoC cluster against a more traditional implementation.

To test the performance of the ARM cluster, we use the ZedBoard cluster fully connected to each other through a Netgear FS524 1Gbps network switch [27] to implement the ARM cluster. Each ZedBoard
has a dual-core ARM Cortex-A9 processor that runs the 32-bit Linaro Ubuntu OS with 256MB of DDR3 RAM. Since each ZedBoard has a dual-core ARM processor, we limit the software application to run two software nodes per board running at 667MHz. Since each software node consists of one computation thread and one handler thread, by limiting the number of software nodes, we reduce the latency added because of the context switch. Algorithm 1 is used to implement the Jacobi method for the software-only system. The ARM processors in the Zynq SoC contain the NEON SIMD extension as part of the ARM core that is optimized for parallel operations such as vector addition, which can give the ARM a performance boost for Jacobi implementation [28]. For our purposes, we do not use the extension, since the goal of the Jacobi method use case is to test the functionality of the API, and mainly calibrate our accelerator architecture with ARM and x86 results.

To test the performance of the hardware nodes, we also use the ZedBoard cluster, fully connected to each other through a 1GBps network switch to implement the ARM-FPGA SoC cluster. We placed two hardware nodes on each ZedBoard, which is the maximum number of hardware nodes that can be placed on each Zynq SoC running at 100MHz. Each hardware node is provided with up to 128 MB of DDR3 memory to be used as the PGAS partition. Two background threads in the software are run to maintain the link between the on-chip network and the off-chip network on the ARM processors of all the boards. The background threads sleep until data are ready to be transferred between the on-chip and the off-chip network. We also run a software control node on the ARM processors that acts as a "master" node responsible for programming all the xPAMS blocks, and printing out the convergence results.

The Intel i5-4440 quad-core processor running at 3.1GHz and 8GB DDR3 RAM is used as a baseline for the ARM cluster and the ARM-FPGA SoC cluster. Algorithm 1 that implements the Jacobi method using the PGAS model was run on the i5 processor with two parallel nodes (four threads) since it provides the best performance on that processor. As we increase the number of nodes on the i5 processor, the number of threads increases as well leading to significant time being spent at the barrier state because of the context-switch.

5.2.5 System Performance

Figure 5.5 shows the runtime results of the Jacobi method for the heat transfer equation for constant number of iterations (1024). The number of parallel nodes for the ARM cluster and the ARM-FPGA cluster is set to eight nodes. The number of parallel nodes for the i5-4440 implementation is set to two nodes. The time shown in the y-axis represents the time taken to run the all iterations of the Jacobi
Figure 5.5: Surface Length vs. Runtime Results for 1024 iterations of the Jacobi Method

Figure 5.6: Breakdown of the Runtime of Jacobi method for 1024 Iterations for the ARM Cluster, the ARM-FPGA Cluster, and the Intel i5 Processor on a 4096x4096 Surface
method. It ignores the time taken to setup the library and the background threads, and programming all the xPAMS blocks. The same holds true for all the experiments in this section. The x-axis represents the total length of the surface that the heat transfer is being measured over. The figure also provides the equation that represents the trendline, and the $R^2$ value that shows how well the trendline fits the plot (the value of one being a perfect fit).

Figure 5.6 shows the breakdown of the runtime of the Jacobi method for a 4096x4096 surface. The computation time comprises the time spent in the computation part of the code as well as the time spent in the second barrier waiting for all the nodes to finish the computation. The communication time comprises the time spent either in the off-chip network (the ARM cluster and the ARM-FPGA cluster), the on-chip network (the ARM-FPGA cluster), or in the memcpy() function (the i5 processor and the ARM cluster for local transfers). The x-axis of the figure uses a log scale to emphasize the communication time.

While the ARM cluster performs relatively fast for the small surface lengths, as the surface length increases, the runtime increases significantly with the majority of the time spent in the computation phase. Based on Figure 5.6 we can see that for a 4096x4096 surface, a significant portion of the runtime is spent in the computation phase, while the communication phase plays a small role in the runtime of the application.

The i5 processor performs significantly better compared to the ARM cluster for larger surface sizes (>256x256). While both the ARM cluster and the i5 processor grow polynomially, the i5 processor performs at roughly 9-fold faster than the ARM cluster for larger surface sizes. Since the Jacobi method on the i5 processor uses only two parallel nodes on the same device, there is no off-chip communication, and very little local communication through memcpy(). Therefore, the share of the communication time in the total runtime of the application is very small (0.25%) as shown in Figure 5.6.

By off-loading the compute intensive task to the hardware nodes, the ARM-FPGA cluster provides significant performance gains for the Jacobi method. While the runtime of the ARM-FPGA also increases polynomially as the surface length increases, the growth is much slower than the i5 processor or the ARM cluster. The ratio of the runtime for the ARM-FPGA cluster and the i5 processor is 0.91, 1.89, and 2.45 for the surface lengths of 1024, 2048, and 4096, respectively. However, running the cluster on a SoC board with larger memory will allow us to test the performance of the ARM-FPGA cluster with larger surfaces, thereby providing a much better performance calibration. The breakdown of the ARM-FPGA cluster runtime for a 4096x4096 surface in Figure 5.6 shows that the computation in the FPGA is about 2.78-fold faster than the computation in the i5 processor, and 25-fold faster than the computation in the ARM cluster. While both the ARM cluster and the ARM-FPGA cluster use the
off-chip communication for remote transfers, the on-chip communication, which takes 39 cycles for a two-way ping-pong, is significantly faster than the \texttt{memcpy()} resulting in slightly less time spent performing the communication.

Figure 5.7 shows the result of a scalability experiment performed to study the effects of increasing the number of parallel nodes. The experiment shows the results of the Jacobi method implemented using THe\_GASNet Extended API over a constant number of iterations (1024). The experiment increases the number of boards, thereby increasing the parallel hardware nodes running the application. The number nodes per board are still kept at two hardware nodes because of resource constraints of the Zynq SoC. The software application is used to provide the background threads for maintaining the on-chip network and off-chip network connection, and the control node to program all the xPAMS. The figure also allows us to compare the performance of an i5 processor with the ZedBoards. For the Jacobi method application, four hardware nodes on two ZedBoards can easily outpace the best performance provided by the i5 processor. However, if the four hardware nodes are placed on one ZedBoard, then we can eliminate the high-latency added by the off-chip communication. It is likely that a single ZedBoard with four hardware nodes can outpace the i5 processor.

As we increase the number of nodes in the cluster, each parallel hardware node works on a smaller subset of the surface, thereby reducing the computation time of the design. Going from two hardware nodes to four hardware nodes, we get roughly two-fold speedup. The speedup results match the expected value, since in the four-node system, each node performs the computation on half the surface area.
Figure 5.8: Performance of the PAMS vs. the xPAMS for 1024 Iterations of the Jacobi Method using Eight Hardware Nodes.

compared to the two-node system (the surface is divided four ways). However, as we keep increasing the number of hardware nodes, the speedup decreases. Going from eight hardware nodes to 16 hardware nodes, we only get 1.5-fold speedup. This can be attributed to the increase in the number of messages being passed as we increase the number of hardware nodes. The major source of the increase in the communication time is the AXI-FSL DMA, which is solely responsible for transferring all the data between the on-chip network and the off-chip network on each board, creating a bottleneck in the dataflow. This bottleneck can be partially solved by using other Zynq-based boards that connect a PL region directly to the Ethernet MAC, thereby removing the need for the AXI-FSL DMA to manage all the transfers to the PL.

For a 4096x4096 surface implemented on eight hardware nodes (two on each ZedBoard), each node runs the Jacobi method on a 1024x2048 surface. The AXI-FSL DMA performs three read operations (2x1024 words, and 1x2048 words) and three write operations of the same size per iteration. Based on the latency values of the AXI-FSL DMA in Appendix C, it can be estimated that a direct hardware-to-off-chip connection will reduce the total runtime by 1.42 seconds for the edge transfers, which gives us 1.23-fold speedup in communication time in Figure 5.6.

A major change introduced to the hardware node is the modification of the PAMS block to support the extended API functions and to provide handler function execution based on an interrupt. Figure 5.8 shows the difference between the runtime performance of the Jacobi method implemented on eight nodes.
Chapter 5. Results and Analysis

It can be observed that the original PAMS performs faster than the xPAMS over all surface lengths. Two factors that are responsible for the slowdown of the xPAMS are: a) the use of extended functions, since all the outgoing requests generate a reply, and b) each incoming message interrupts the xPAMS, and takes a minimum of three cycles to resolve. While the xPAMS slowdown is constant for all iterations of the Jacobi method, more applications will be implemented in the future to accurately characterize xPAMS performance. However, the trade-off of the reduced performance is increased ease of programming.

Figure 5.9 shows the result of the experiment set to determine if any significant overhead incurred when the Jacobi method is run using the THe_GASNet Extended API functions vs. the THe_GASNet Core API functions for the software implementation of the Jacobi method on the ARM Cluster. Note that to pass the convergence results back to the control node in the software application, a short AM per iteration is still used. However, since it only carries one 32-bit number as an argument, the effect is negligible. The experiment runs the Jacobi method for 1024 iterations on eight parallel nodes. The software application performs the same computation tasks, but uses either the core API functions, the non-blocking memory transfer functions with explicit handles, or the non-blocking memory transfer function with implicit handles, for message passing.

The y-axis and the x-axis use the logarithmic scale to emphasize the difference between the THe_GASNet Extended API results and the THe_GASNet Core API results for small data sizes. Similar to the xPAMS results, the extended API performs slower than the core API. For the larger surface lengths, the runtime
between the core API and the extended API looks the same because the large computation time masks the differences in the communication time. For the smaller surface lengths, such as 32x32 and 64x64, where the computation time is small enough to notice the change in the communication time, it can be observed that the extended API implementation performs about 1.24-fold slower than the core API implementation. The slowdown in the extended software library is expected since it is a higher-level library that enforces the reply message for each request message increasing the communication time. Since both the implicit handling and the explicit handling use the same request handler and the reply handler functions, there is no difference between the runtime results when using either of the extended API options.

5.2.6 Resource Utilization

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>Flip Flops</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAMS</td>
<td>2383</td>
<td>1151</td>
<td>1</td>
</tr>
<tr>
<td>xPAMS</td>
<td>3048</td>
<td>1344</td>
<td>2</td>
</tr>
<tr>
<td>Overall</td>
<td>20,000(37%)</td>
<td>14,188(13%)</td>
<td>75(54%)</td>
</tr>
</tbody>
</table>

Table 5.2: Utilization of the PL Region in the Zynq SoC for Two Hardware Nodes

Table 5.2 shows the resource utilization by the PAMS and the xPAMS block. Making the PAMS/xPAMS as a low-resource block comes at the expense of the support for general operations. By implementing more applications on the platform, the limitations of the xPAMS can be accurately accessed. The table also shows the overall FPGA utilization when implementing the design shown in Figure 3.1. Compared to the available resources, the usage of the overall design is very low, which is an ideal case for us since it allows us to increase the number of nodes per board. However, the placement and routing tool currently fails to route the design with more than two parallel nodes because it cannot meet the timing. Optimizing the design may allow us to fix the timing issues and test the application with more parallel nodes.
Chapter 6

Conclusions

In this thesis we present the architecture that was used to implement THe\_GASNet Extended API on a Zynq SoC cluster. We also developed the high-level communication library that provides the PGAS programming model support for the hardware accelerators on the cluster. The programming model implemented allows the software programmer to treat the hardware nodes as another parallel processor that is, ideally, indistinguishable from a software processor.

The project’s main goal is to provide a high-level communication library that can be leveraged by a PGAS runtime compiler to control the hardware nodes, reducing the need to involve a user when migrating an application to the FPGA for higher performance. The THe\_GASNet Core API, developed as part of previous research, provides a uniform message-passing layer between the hardware nodes and the software nodes, thereby enabling a programmer to implement parallel applications in the FPGA without needing to focus on implementing the communication aspect in hardware. Our work ported the THe\_GASNet Core API to the Zynq SoC environment by developing device-specific networks for the core API in line with the suggestion of the GASNet API developers. A higher-level library was developed based on the core API to allow the programming of applications developed in other PGAS libraries easier. The higher-level library also serves as a bridge between the core API and a PGAS runtime language, such as UPC, which is implemented on top of the GASNet API.

Having a standard PGAS programming API for parallel applications on FPGA reduces the time to port other HPC applications to the platform. Our work provides a standard interface that allows a programmer to control the accelerators, perform memory operations, and use complex communication patterns through high-level functions further reducing the time spent to develop applications on the FPGA. The majority of the work involved when porting new applications to the current infrastruc-
ture now can be focused on developing the core, rather than developing memory and communication interfaces.

We used a Zync SoC cluster to implement the Jacobi method to solve the Heat Transfer Equation to test the performance of the FPGA and identify the overhead of the high-level functions. The THe_GASNet Extended API performs slower than the original THe_GASNet Core API, with the slowest performance of 80% of the core API runtime at small surface lengths ($<1024 \times 1024$). However, the extended API provides greater programmability when compared to the core message-passing functions, which is demonstrated by a code comparison. Since all the off-chip communication between hardware nodes has to go through the ARM processors on the ZedBoard, it introduces significant latency. Despite the handicap, two ZedBoards outpaced the best performance achieved using the Intel i5-4440 processor for our application. Using four ZedBoards (eight hardware nodes), the application achieves the speedup of 2.78-fold over the i5 implementation. However, the ZedBoard does not have provisions for power analysis, which is another key metric to compare the i5-4440 processor with the SoC implementation.

To quantify the performance of the individual components of the architecture, we implemented various tests to measure the performance of the on-chip network, off-chip network, and the PS-PL communication. The results from these tests helped explain the performance of the Jacobi method, and provide areas of improvement and further research.

This work has shown that it is feasible to use FPGAs in the PGAS programming model to achieve acceleration. It has also shown that even a somewhat handicapped implementation using an ARM-FPGA platform can be competitive with a more typical x86 platforms suggesting that such SoC platforms may have a role in data centers and HPC.

6.1 Future Work

This thesis validates the functionality of the THe_GASNet Extended API on the ARM-FPGA SoC cluster. This section suggests some short-term and long-term work to further enhance the capabilities of the high-level library, and the accelerator integration to the PGAS programming model.

Currently, we are using the Xilinx ISE 14.7 tools to develop the architecture because some of the hardware IPs used in our system are not supported by Xilinx Vivado Design Suite. However, since Xilinx has stopped the support for ISE for the future chip generations, one of the most immediate needs is for the project to be moved to Vivado. This means that the we have to use newer versions of the NetIf blocks that are designed to use the AXI4 Stream Channels instead of the FSL. This also allows us to use the high-performance AXI DMA block instead of the AXI-FSL DMA, which is a bottleneck in our
The migration to Vivado allows us to move the architecture to use more advanced Zynq SoCs, such as Xilinx XC7Z045/XC7Z100-2FFG900 [21] on the Zynq Mini-ITX Development Board [29], which provides 2GB of DDR memory that can be used to test much larger applications. More importantly, the Mini-ITX board provides Ethernet support from the PL region that can be used to more directly connect the on-chip networks in each FPGA. This will help alleviate the communication overhead of going through the ARM processors to the off-chip network.

While the current library meets the needs of the Jacobi method application, more applications will allow us to test the re-usability of the library and find areas where more support needs to be added. Currently the support of the high-level functions is limited because of the xPAMS block. Making the xPAMS support all the functionalities of the library will significantly increase the size of the xPAMS, so a trade-off between size and functionality should be tested.

Currently, the hardware accelerator cores are hand-designed, however, using the HLS tool to generate the compute intensive sections of the code is a possibility. To successfully generate the cores, either the generated cores have to provide the interface that is currently used by the xPAMS, or the xPAMS interface has to be changed to a standard interface that is already supported by the HLS.

While both the THe_GASNet Core API and the THe_GASNet Extended API provide communication functions, they are meant to be used as a communication library. Therefore, support for a PGAS language such as UPC should be added to make the application development faster for the FPGAs.

The end goal is to provide a plug and play model where the accelerator cores can be generated from the HLS tools, and plugged into an existing network of THe_GASNet Supporting IPs. Simultaneously, the same application code can be used by a PGAS runtime compiler to generate the xPAMS code and program the xPAMS, thereby removing the need for the user to go through long design cycles for generating the accelerator cores or programming the xPAMS.
Appendices
Appendix A

THe_GASNet Core API Message Format

This appendix contains the format of the short, medium, and long message types as they are passed between the parallel nodes. The format of the messages remain the same regardless of the type of the node use, there creating a uniform message format that allows message passing between hardware nodes, software nodes, and MicroBlaze nodes. Sections below were provided by Ruediger Willenberg, and forms the message passing layer of the THe_GASNet Core API.

A.1 Active Message Headers

This sections shows the formats of Active Message headers as used between all platforms. All Long-type headers are followed by the payload.

All types of messages have an initial 8-bit word specifying the type of Active message:

- Bit 0: Set if Short or Long Strided message
- Bit 1: Set if Medium or Long Vectored message

<table>
<thead>
<tr>
<th>Contents</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
<td>Short, Medium, Long</td>
</tr>
<tr>
<td>#Arguments (8)</td>
<td>Medium and Long only</td>
</tr>
<tr>
<td>Handler ID (16)</td>
<td>Long only</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td></td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
<td>Long only</td>
</tr>
</tbody>
</table>

Table A.1: Active Message parameters - Standard types
• Bit 2: Set if any type of Long message

• Bit 4: Set if data payload for a Medium or Long message is supplied by a FIFO instead of from memory (only applicable to FPGA components)

• Bit 5: Set if a Medium or Long message is of type Async, meaning a function call returns without explicit confirmation of a completed memory read

• Bit 6: Set if message is of Reply type, otherwise it is of Request type.

• Bits 3 and 7: Reserved

The same word also holds the identifier of the handler function to be called on arrival, and the number of attached handler function arguments. The next word specifies the payload size in bytes for Medium and Long messages. In case of a Long message, this is followed by a 64-bit destination address for the payload.

<table>
<thead>
<tr>
<th>Contents</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
<td></td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td></td>
</tr>
<tr>
<td>Contiguous block size in bytes (32)</td>
<td></td>
</tr>
<tr>
<td>Number of contiguous blocks (32)</td>
<td></td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
<td></td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
<td></td>
</tr>
<tr>
<td>Stride in bytes between contiguous blocks (32)</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Active Message parameters - Strided Long type

A.2 Active Message Request Headers

This sections shows the request formats as transmitted from MicroBlaze or computation core to GAScore. All headers with the FIFO bit set are followed by the payload data.

A.3 Active Message Request Headers

This sections shows the formats of handler requests as transmitted from GAScore to MicroBlaze or computation core. All headers of Medium-type are followed by the payload data.
### Table A.3: Active Message request parameters - Standard types

<table>
<thead>
<tr>
<th>Contents</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
<td>Source (16)</td>
</tr>
<tr>
<td>Source address, lower half (32)</td>
<td>Medium, Long (FIFO=0)</td>
</tr>
<tr>
<td>Source address, upper half (32)</td>
<td>Medium, Long (FIFO=0)</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
<td>Short, Medium, Long</td>
</tr>
<tr>
<td>0..512 words memory payload (32)</td>
<td>Medium, Long (FIFO=1)</td>
</tr>
</tbody>
</table>

### Table A.4: Active Message request parameters - Strided Long type

<table>
<thead>
<tr>
<th>Contents</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td>Short, Medium, Long</td>
</tr>
<tr>
<td>Source address, lower half (32)</td>
<td>Medium and Long only</td>
</tr>
<tr>
<td>Source address, upper half (32)</td>
<td>Medium and Long only</td>
</tr>
<tr>
<td>Source: Stride in bytes between contiguous blocks (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Source: Contiguous block size in bytes (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Source: Number of contiguous blocks (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination: Stride in bytes between contiguous blocks (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination: Contiguous block size in bytes (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Destination: Number of contiguous blocks (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
<td>Long only</td>
</tr>
</tbody>
</table>

### Table A.5: Handler function call parameters - Standard types

<table>
<thead>
<tr>
<th>Contents</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td>Medium and Long only</td>
</tr>
<tr>
<td>Start address, lower half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Start address, upper half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
<td>Medium only</td>
</tr>
<tr>
<td>0..512 words memory payload (32)</td>
<td>Medium only</td>
</tr>
</tbody>
</table>

Table A.3: Active Message request parameters - Standard types

Table A.4: Active Message request parameters - Strided Long type

Table A.5: Handler function call parameters - Standard types
Appendix A. The GASNet Core API Message Format

Contents

<table>
<thead>
<tr>
<th>AM type (8)</th>
<th>#Arguments (8)</th>
<th>Handler ID (16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contiguous block size in bytes (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of contiguous blocks (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start address, lower half (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start address, upper half (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stride in bytes between contiguous blocks (32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.6: Handler function call parameters - Strided Long type
Appendix B

xPAMS OPCODES

This section lists the low level codes used by the software control node to program the xPAMS. Note that high-level functions for xPAMS corresponding to the extended API have been created to make the programming of the xPAMS easier. It may make it easier for a PGAS compiler read a software application, and generate a corresponding xPAMS code.
## Appendix B: xPAMS Opcodes

### Figure B.1: Low-level PAMS/xPAMS instruction set

<table>
<thead>
<tr>
<th>hex</th>
<th>Mnemonic</th>
<th>Opcode*</th>
<th>r1</th>
<th>r0</th>
<th>EndIA</th>
<th>Source</th>
<th>Description</th>
<th>Parameters in opcode</th>
<th>Implemented</th>
<th>Extra words</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PAMS_WANTFOR (PAMS_WP)</td>
<td>0b000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Wait for masked</td>
<td>Get (16) / Pull (16) / Ctrl inputs (48) / MagQCtrl (48) / TmsrCtrl (48) / Timer (25) / All return (18) / Code loaded (18)</td>
<td>Y</td>
<td>Words (s)</td>
<td></td>
</tr>
<tr>
<td>04-09B</td>
<td>PAMS_TO_ASENDB</td>
<td>0b000001</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>Write to AM send pipe from source</td>
<td>- (48) / #Words (128)</td>
<td>Y</td>
<td>Words (s)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>PAMS_BRANCH</td>
<td>0b001010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Branch absolute</td>
<td>Flag bits (48) / Address (128)</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>PAMS_SETNODE</td>
<td>0b010101</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Set node X</td>
<td>Node X (18)</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>PAMS_ADV TIMER</td>
<td>0b011010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Correct timer offset</td>
<td>-</td>
<td>Y</td>
<td>Offset (16)</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>PAMS_TRESH</td>
<td>0b011010</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Set timer threshold</td>
<td>- (16) / Threshold mask (28)</td>
<td>Y</td>
<td>Value (16)</td>
<td></td>
</tr>
<tr>
<td>2E</td>
<td>PAMS_TXH</td>
<td>0b011011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Set Message counter threshold</td>
<td>Handler Code (16) / Counter mask (48)</td>
<td>Y</td>
<td>Value (16)</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>PAMS_TXOUT</td>
<td>0b011000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pulse Ctrl Output</td>
<td>Output mask (32)</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>PAMS_TO_CORE</td>
<td>0b011000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>Feed data to core</td>
<td>- (48) / #Words (128)</td>
<td>Y</td>
<td>Words (s)</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>PAMS_RESET</td>
<td>0b101011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Reset wait flags</td>
<td>Ctrl inputs (48) / MagQCtrl (48) / TmsrCtrl (48) / Timer (25)</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>PAMS_BREAK</td>
<td>0b111110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Return from handler</td>
<td>-</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other possible ops:
- Accumulate immediate (16 bits) (Better latency)

Accumulator ops:
- All message handler codes

<table>
<thead>
<tr>
<th>00</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Acc &lt; Opword</td>
</tr>
<tr>
<td>02</td>
<td>Acc = Opword</td>
</tr>
<tr>
<td>03</td>
<td>Acc AND Opword</td>
</tr>
</tbody>
</table>

| 04 | New Code to run at address 0 (start?) |
| 05 | New Code to return after current code (e.g., procedure) |
| 06 | New Code to put at address 0[28] |
| 07 | New Code to put at address 0[28] (e.g., set new operation) |
| 08 | Run the handler for a get request |
| 09 | Run the handler for a get reply |
| 0A | Run the handler for a put request |
| 0B | Run the handler mandated to OP COMPLETE the explicit handles |
| 0C | Run the handler for a get request (e.g., procedure) |
| 24 | DirectReply. Short Messages whose arguments are found right back to serve as remote reads |
| 25 | Don't save arrival time => follow up with sending last arrival time back |
Appendix C

AXI-FSL DMA Characteristics

This appendix lists the results of the throughput and latency test on the AXI-FSL DMA.
### Table C.1: DMA Write Throughput and Latency

<table>
<thead>
<tr>
<th>Data Size (Bytes)</th>
<th>Throughput (MB/s)</th>
<th>Latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.86</td>
<td>0.000037</td>
</tr>
<tr>
<td>64</td>
<td>4.57</td>
<td>0.000014</td>
</tr>
<tr>
<td>128</td>
<td>9.41</td>
<td>0.0000136</td>
</tr>
<tr>
<td>256</td>
<td>6.12</td>
<td>0.0000418</td>
</tr>
<tr>
<td>512</td>
<td>11.03</td>
<td>0.0000464</td>
</tr>
<tr>
<td>1024</td>
<td>38.79</td>
<td>0.000026</td>
</tr>
<tr>
<td>2048</td>
<td>78.77</td>
<td>0.000026</td>
</tr>
<tr>
<td>4096</td>
<td>123.37</td>
<td>0.0000332</td>
</tr>
<tr>
<td>8192</td>
<td>165.83</td>
<td>0.0000494</td>
</tr>
<tr>
<td>16384</td>
<td>147.07</td>
<td>0.0001114</td>
</tr>
<tr>
<td>32768</td>
<td>138.26</td>
<td>0.000237</td>
</tr>
<tr>
<td>65536</td>
<td>134.35</td>
<td>0.0004878</td>
</tr>
<tr>
<td>131072</td>
<td>127.58</td>
<td>0.0010274</td>
</tr>
<tr>
<td>262144</td>
<td>134.90</td>
<td>0.0019432</td>
</tr>
<tr>
<td>524288</td>
<td>134.30</td>
<td>0.0039038</td>
</tr>
<tr>
<td>1048576</td>
<td>130.37</td>
<td>0.0080428</td>
</tr>
<tr>
<td>2097152</td>
<td>130.85</td>
<td>0.016027</td>
</tr>
<tr>
<td>4194304</td>
<td>129.90</td>
<td>0.0322876</td>
</tr>
<tr>
<td>8388608</td>
<td>131.17</td>
<td>0.0639522</td>
</tr>
</tbody>
</table>

### Table C.2: DMA Read Throughput and Latency

<table>
<thead>
<tr>
<th>Data Size (Bytes)</th>
<th>Throughput (MB/s)</th>
<th>Latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.66</td>
<td>0.000048</td>
</tr>
<tr>
<td>64</td>
<td>1.34</td>
<td>0.000048</td>
</tr>
<tr>
<td>128</td>
<td>2.66</td>
<td>0.000048</td>
</tr>
<tr>
<td>256</td>
<td>5.16</td>
<td>0.000050</td>
</tr>
<tr>
<td>512</td>
<td>9.59</td>
<td>0.000053</td>
</tr>
<tr>
<td>1024</td>
<td>21.60</td>
<td>0.000047</td>
</tr>
<tr>
<td>2048</td>
<td>37.79</td>
<td>0.000054</td>
</tr>
<tr>
<td>4096</td>
<td>61.13</td>
<td>0.000067</td>
</tr>
<tr>
<td>8192</td>
<td>89.82</td>
<td>0.000091</td>
</tr>
<tr>
<td>16384</td>
<td>103.17</td>
<td>0.000159</td>
</tr>
<tr>
<td>32768</td>
<td>113.07</td>
<td>0.000290</td>
</tr>
<tr>
<td>65536</td>
<td>118.38</td>
<td>0.000554</td>
</tr>
<tr>
<td>131072</td>
<td>121.79</td>
<td>0.001076</td>
</tr>
<tr>
<td>262144</td>
<td>123.25</td>
<td>0.002127</td>
</tr>
<tr>
<td>524288</td>
<td>106.29</td>
<td>0.004933</td>
</tr>
<tr>
<td>1048576</td>
<td>114.27</td>
<td>0.009176</td>
</tr>
<tr>
<td>2097152</td>
<td>117.10</td>
<td>0.017908</td>
</tr>
<tr>
<td>4194304</td>
<td>118.79</td>
<td>0.035309</td>
</tr>
<tr>
<td>8388608</td>
<td>116.50</td>
<td>0.072003</td>
</tr>
</tbody>
</table>
Bibliography


