HETEROGENEOUS RUNTIME SUPPORT FOR
PARTITIONED GLOBAL ADDRESS SPACE PROGRAMMING ON FPGAS

by

Ruediger Willenberg

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Ruediger Willenberg
We are presenting THeGASNet, a framework to provide remote memory communication and synchronization in heterogeneous, distributed systems composed of software components and FPGA components. It is intended as a runtime layer to support higher-level languages and libraries that implement the Partitioned Global Address Space (PGAS) model. PGAS is a shared memory parallel programming model intended for high-productivity programming of distributed, cluster-like systems. THeGASNet provides a communication abstraction with a common API for both software and hardware components, thereby facilitating easier migration of performance-critical application portions to reconfigurable computing hardware.

To demonstrate the development flow, we have implemented three applications representing common distributed application characteristics, starting with software-only solutions and using the common API to efficiently move selected parts into FPGA hardware. Based on the accumulated experience, we illustrate why PGAS is a good model to program heterogeneous systems using FPGAs, define minimum infrastructure requirements and outline a vision for continued exploration of heterogeneous PGAS programming.
Dedication

Dedicated to the Fellows of Massey College and Master Emeritus John Fraser,
for exemplifying all that is great about Canada and its wonderful people,
and for giving me a home away from home.
Acknowledgements

Many people have supported me in finishing this work in tangible and intangible ways. First and foremost, I owe a great debt to my academic supervisor, Paul Chow, for giving good advice and encouragement, providing generous financial support beyond his initial commitment, emanating serenity and good cheer, owning a seemingly endless reservoir of patience, assembling an excellent group of fellow graduate students and most importantly, taking a chance on me and kicking off one of the happiest and most inspiring chapters of my life.

Many thanks go to my supervisory committee for encouragement and good advice: Tarek Abdelrahman, the late Greg Steffan, who I remember as kind, supportive and easygoing, and Vaughn Betz, who kindly stepped in after Greg’s passing.

All this would have been so much harder without a great group of fellow grad students and lab mates, in no particular order and undoubtedly incomplete: Jasmina Vasiljevic, Vincent Mirian, Xander Chin, Fernando Martin del Campo, Charles Lo, Sanket Pandit, Stuart Byma, Jimmy Lin, Ehsan Ghasemi, Jeff Zhang, Naif Taradar, Rob Hesse, Bill Teng, Andrew Shorten, Tahir Diop, Marcel Gort, Jin Hee Kim, Joy Chen, Steven Gurfinkel, Mario Badr and Eric LaForest.

This work would not have been possible without the excellent inspiration and groundwork of Manuel Saldaña’s TMD-MPI project, which originally got me interested in working with Paul. Sanket Pandit and Muhammad Khurram Tariq have contributed tremendously by enabling the framework’s portability to ARM.

Implementing our system on the $BEE^{4}$ was challenging work, and I could not have succeeded without support by Kevin Camera of $BEEcube$ and Hugh Pollitt-Smith and Yassine Hariri of $CMC$ Microsystems. Many thanks to Xilinx, $CMC$ Microsystems and emSYSCAN for funding and free design software. I am also grateful to Sigasi nv and especially Philippe Faes for providing me with free VHDL design software as well as Xillybus ltd. for free use of their PCIe core.

The communities of Massey College and Knox College made my time in Canada so much more than just graduate studies and guaranteed that I now feel homesick on two continents... Thank you to Jonathan Rose for pointing me towards Massey.

I am very grateful to the University of Applied Sciences, Mannheim and its Department of Information Technology for giving me a job I love, and the time to finish this degree so I can keep it.

Last but absolutely not least, everything was made easier by enjoying transatlantic love and support from Mama, Papa, Britta, Marc, Lasse and Moritz.
# Contents

1 Introduction 1
   1.1 Motivation .............................................. 1
   1.2 Unified Programming Model ................................. 3
   1.3 Research goals .......................................... 7
   1.4 Thesis organization .................................... 9

2 Background 10
   2.1 Forms of parallelism ................................... 10
   2.2 Distributed parallel programming models ................. 12
      2.2.1 Shared memory model .............................. 12
      2.2.2 Distributed memory model ......................... 13
      2.2.3 The Partitioned Global Address Space model ..... 14
   2.3 Related Work ........................................... 15
      2.3.1 PGAS implementations .............................. 15
      2.3.2 Previous work on PGAS and FPGAs ................. 17
      2.3.3 Other related work ............................... 18

3 THeGASNet - A heterogeneous GASNet implementation 20
   3.1 Structure of a generic PGAS stack ....................... 20
   3.2 Choosing a communications API for heterogeneous systems 22
   3.3 The GASNet API and library ............................ 24
      3.3.1 Active Messages and Handler functions ............ 25
      3.3.2 Sequence of GASNet node operations .............. 29
3.3.3 GASNet Core API: code example ........................................... 31
3.4 Enabling cross-platform compatibility for THeGASNet ...................... 37
  3.4.1 Target architectures ................................................ 37
  3.4.2 Interoperability issues .............................................. 39
  3.4.3 Packet format ...................................................... 40
3.5 THeGASNet software library for regular CPUs .............................. 41
  3.5.1 Implementation strategy ........................................... 41
  3.5.2 Linux-based multi-threaded implementation ............................... 43
  3.5.3 Differences to the GASNet Core API specification .......................... 46
  3.5.4 Platform-independent Non-GASNet functionality ........................... 49
  3.5.5 THeGASNet code example ......................................... 50

4 THeGASNet implementation in the FPGA fabric ............................ 55
  4.1 MicroBlaze THeGASNet software implementation .......................... 57
  4.2 THeGASNet hardware components ..................................... 59
    4.2.1 NetIf communication ........................................... 59
    4.2.2 GAScore: Remote DMA support .................................. 61
    4.2.3 Programmable Active Message Sequencer ............................ 71
    4.2.4 Code example: THeGASNet with PAMS code generation ............... 75

5 Implementation platforms .................................................. 80
  5.1 MapleHoney Reconfigurable Computing Cluster ............................ 80
    5.1.1 BEEcube BEE4 .................................................. 80
    5.1.2 Platform-specific components .................................... 82
    5.1.3 Synthetic communication benchmarks for the MapleHoney platform ... 87
  5.2 Digilent Zedboard .................................................. 92
  5.3 Portability to other platforms .................................... 94

6 Application use cases .................................................... 95
  6.1 Jacobi Relaxation of Electrostatic Differential Equation .................... 96
    6.1.1 Software-only implementation .................................. 99
6.1.2 Isolation of accelerator code and hardware modeling . . . . . . . . . . . . . . 101
6.1.3 Jacobi hardware core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.2 Breadth-First Search (Graph500) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
   6.2.1 Software-only implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 107
   6.2.2 Isolation of accelerator code and hardware modeling . . . . . . . . . . . . . . 111
   6.2.3 BFS hardware core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
6.3 2D-DFT and stream encryption (ChaCha20) . . . . . . . . . . . . . . . . . . . . . . . 116
   6.3.1 Software-only implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 116
   6.3.2 Isolation of accelerator code and hardware modeling . . . . . . . . . . . . . . 121
   6.3.3 Hardware cores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.4 Application performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
   6.4.1 Jacobi relaxation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
   6.4.2 Breadth First Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
   6.4.3 2D-FFT and encryption . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

7 Why PGAS is a good fit for distributed FPGA systems . . . . . . . . . . . . . . . . . . . 130
   7.1 Communication and cost exposure . . . . . . . . . . . . . . . . . . . . . . . . . . 131
   7.2 Synchronization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
   7.3 Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
   7.4 Task partitioning and migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
   7.5 Productivity gains vs. performance overhead for FPGA components . . . . . . . 135
       7.5.1 Productivity and overhead in comparison with TMD-MPI . . . . . . . . . . . 139

8 Integrating FPGAs into PGAS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
   8.1 FPGA properties and resulting requirements . . . . . . . . . . . . . . . . . . . . . . . 141
       8.1.1 Memory management . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
       8.1.2 Separation of data flow and high-level control flow . . . . . . . . . . . . . . 142
       8.1.3 Application launch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
       8.1.4 Modeling heterogeneity at the application level . . . . . . . . . . . . . . . . 145
   8.2 PGAS requirements for forthcoming FPGA platforms . . . . . . . . . . . . . . . . . . 146
List of Tables

3.1 Active Message header parameters - *Long type* ............................................ 41
3.2 Differences between original GASNet and THeGASNet ........................................ 49

4.1 Post-map resource utilization for various *NetIf* configurations ............................... 61
4.2 Active Message request parameters - *Long message* ........................................... 64
4.3 Handler function call parameters - *Short message* .............................................. 66
4.4 Post-map resource utilization for *GAScore* ....................................................... 70
4.5 Post-map resource utilization, *PAMS vs. MicroBlaze* .......................................... 75

5.1 Post-map resource utilization of memory controller and multi-port AXI interface .......................... 85
5.2 Post-map resource utilization for PCIe core with *NetIf* interface ............................... 86
5.3 Post-map resource utilization for *fsl2ic* off-chip communication link .......................... 86
5.4 Latency for a one-way *Short Active Message* .................................................... 88
5.5 Latency for a two-way *Short Active Message bounce* ........................................... 88
5.6 Latency to complete a barrier .............................................................................. 89
5.7 Latency for memory copies (*Long Active Messages*) inside a single FPGA ..................... 89
5.8 Sustained transfer bandwidth(*Long Active Messages*) inside an FPGA ...................... 90
5.9 “*Flooding*” - sustained all-to-all communication .................................................. 90
5.10 Latency for a two-way *Short Active Message communication* ................................. 90
5.11 Sustained transfer bandwidth(*Long Active Messages*) over PCIe ............................... 91
5.12 Latency for a two-way *Short Active Message communication* ................................. 91
5.13 “*Flooding*” - sustained all-to-all communication .................................................. 92

6.1 Post-map resource utilization for *Jacobi* stencil core ............................................ 104
6.2 Post-map resource utilization for Breath-First Search core . . . . . . . . . . . . . 116
6.3 Post-map resource utilization for Nx1d-FFT and ChaCha20 cores . . . . . . . . 123
6.4 Jacobi relaxation performance, 4096x4096 cells . . . . . . . . . . . . . . . . . . . 125
6.5 Jacobi relaxation performance, 8192x4096 cells . . . . . . . . . . . . . . . . . . . 125
6.6 Jacobi relaxation performance, 8192x8192 cells . . . . . . . . . . . . . . . . . . . 126
6.7 BFS performance at scale 22 (4.2 million vertices) . . . . . . . . . . . . . . . . . 127
6.8 BFS performance at scale 23 (8.4 million vertices) . . . . . . . . . . . . . . . . . 127
6.9 BFS performance at scale 24 (16.7 million vertices) . . . . . . . . . . . . . . . . . 127
6.10 4096x4096-point 2D-FFT + ChaCha20 encryption, all values are msecs . . . . 128

B.1 Active Message parameters - Standard types . . . . . . . . . . . . . . . . . . . . 167
B.2 Active Message parameters - Strided Long type . . . . . . . . . . . . . . . . . . 168
B.3 Active Message parameters - Vectored Long type . . . . . . . . . . . . . . . . . 168
B.4 Active Message request parameters - Standard types . . . . . . . . . . . . . . . 169
B.5 Active Message request parameters - Strided Long type . . . . . . . . . . . . . 169
B.6 Active Message request parameters - Vectored Long type . . . . . . . . . . . . . 170
B.7 Handler function call parameters - Standard types . . . . . . . . . . . . . . . . . 171
B.8 Handler function call parameters - Strided Long type . . . . . . . . . . . . . . . 171
B.9 Handler function call parameters - Vectored Long type . . . . . . . . . . . . . . 172
List of Figures

1.1 Multiple platforms with common parallel API ........................................... 4
1.2 Communication in distributed systems with accelerators .......................... 5

2.1 Distributed parallel programming models .................................................. 12

3.1 General protocol stack of a PGAS application ........................................... 21
3.2 Protocol stack of a GASNet-based PGAS application ................................. 24
3.3 Remote write ..................................................................................... 28
3.4 Remote read ....................................................................................... 29
3.5 Remote string copy example .................................................................... 35
3.6 Host target processors ............................................................................ 37
3.7 MicroBlaze processors on FPGA ............................................................. 38
3.8 Custom computation cores on FPGA ......................................................... 39
3.9 THeGASNet Active Message format ....................................................... 40
3.10 Multi-threaded THeGASNet software implementation ............................. 45

4.1 Host-to-FPGA migration with THeGASNet ............................................... 56
4.2 MicroBlaze-to-Hardware substitution with GAScore .................................. 57
4.3 Typical on-chip NetIf configuration .......................................................... 60
4.4 Overview of GAScore internals ............................................................... 62
4.5 GAScore memory read and Active Message transmission unit .................. 63
4.6 GAScore Active Message reception and memory write unit ..................... 65
4.7 GASNet node with CPU using GAScore .................................................. 68
4.8 GASNet node with custom processing hardware using GAScore ............. 69
Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are a valuable tool to accelerate computation tasks. Since virtually any digital circuit can be implemented on an FPGA, it is possible to freely design pipelines and dataflows tailored to any computation problem. The amount of parallel computation resources provided can be adjusted to match other performance bounds like memory or network bandwidth. Since only the hardware required for a specific problem is instantiated and data movement from and to memory is minimized, FPGAs use less energy per computation than both standard processors (Central Processing Units or CPUs) and Graphics Processing Units (GPUs) [1][2].

Despite these advantages, very few computing systems employ FPGAs. A stark manifestation of this can be observed in the area of High-Performance Computing (HPC), in which the eponymous goal of performance is now increasingly joined by the need to limit power consumption per operation. Both goals should be served well by using FPGAs for suitable problem classes. To the best of our knowledge, however, none of the systems in the current Top 500 list of supercomputers [3] employ FPGAs for computing (FPGAs are used for networking tasks, which are a natural fit for the FPGAs’ dataflow orientation, and which do not change with the computing application).

A major reason for the limited adoption of FPGAs is that they are considered “too hard to program” by conventionally-trained software programmers, which includes scientists from...
many domains that write software to evaluate and simulate their concepts and process their experimental data. This perceived difficulty manifests itself in several obstacles:

1. When designing solutions appropriately structured for FPGAs, these solutions still have to be expressed in fairly low-level 
   *Hardware Description Languages* (HDLs) like VHDL and Verilog, which are somewhat mismatched to the synthesis problem for historical reasons. Rough analogies place these languages at a similar abstraction level as Assembly language or at most as C.

2. Arguably the most intractable problem in FPGA design is the extensive implementation time. Large-scale FPGA circuits can take hours and even whole days to map, place and route. In contrast, incremental software builds usually take seconds or at most a few minutes. Long implementation time does not just lead to a delayed end result, but more importantly, it does not allow the frequent re-assessment of bug fixes and design iterations that is commonplace with software.

3. Popular and commonly taught imperative programming languages like C/C++, Java and Fortran are based on a memory-centric execution model of universal computers built on the work of Turing [4], von Neumann [5], Zuse [6] and other contemporaries. Unfortunately, this is not a good approach to programming FPGAs, as it constrains their major asset, the ability to combine logic and memory into arbitrary, customized dataflows.

4. Lastly, there is no generalized approach on how to include and make use of FPGAs in conventional software designs. For example, there is no unified standard to interface with a host of different FPGA devices and platforms. In contrast, GPUs have been made accessible to a large user community by the OpenCL [7] and CUDA [8] APIs.

In pursuit of the first problem, promising improved HDL idioms like *System Verilog* [11], *Bluespec System Verilog* [12] and *Chisel* [13] have been introduced in recent years. FPGA and tool vendor support for them varies and neither has yet been wholeheartedly embraced by the professional developer community.

\footnote{At the time of publication, both major FPGA vendors have introduced OpenCL adaptations based on High-Level Synthesis for some of their devices [9][10], offering an alternative to some aspects of our proposed solution.}
To solve both the first and the third problem, the two major FPGA vendors have recently focused on High-Level Synthesis (HLS), the description of FPGA hardware through C code. Leaving aside the discussion whether C can be considered High Level by modern standards, using a pervasive software language has opened up access to FPGAs to a larger group of people and enabled easier conversion of existing software code to hardware. The success of such efforts is highly dependent on the kind of code to be translated, and productive use still requires an understanding of how digital systems work. However, suitable problems, especially from the streaming domain, can be solved much more productively due to implicit instantiation of arithmetic units and buffer memories, more flexible abstractions of data types and configurable loop parallelization. Problems not fitting this mold still have to be solved using the traditional hardware description languages or domain-specific solutions where available.

The second problem, long implementation times, has been approached from several directions: Parallelization of the involved algorithms with the onus on maintaining deterministic results [14][15], usage of simpler fine- and coarse-grained overlay architectures that offer faster implementation of digital circuits while maintaining some of the FPGA fabric’s flexibility [16][17], and the usage of partial reconfiguration to minimize re-implementation to the parts of the design that are changing. Our proposed work does not address this problem at all, however future work can make use of our platform abstractions to support partial reconfiguration (see Section 9.1).

The work introduced in this thesis is dedicated to the last problem, the productive integration of FPGA components into software-centric computing systems through a standard interface. In contrast to current FPGA vendor approaches and programming models centered on accelerator integration for single nodes, we are focussing our efforts on a solution that is scalable to large cluster-like distributed systems, since improving FPGA accessibility in this domain is one of our motivations as described above.

1.2 Unified Programming Model

The fundamental idea at the center of our software-oriented approach is a unified programming model and API for all components in a heterogeneous system, as shown in Figure 1.1.
In this approach, the same API is used by application processes running on traditional desktop or server CPUs as well as on embedded CPUs located on the FPGA. While on the traditional CPU-based system, the API is implemented entirely as a software runtime library, on the FPGA, performance-critical functionality can be implemented by hardware components, leaving only a thin API translation layer to interface between software and hardware.

A significant benefit of such a common API is the easy migration of performance-critical application kernels to hardware. An algorithm can be implemented, verified and profiled completely in software, taking advantage of the sophisticated development and debugging tools available in this domain. When profiling has identified the code sections taking up the most execution time, these can be converted into custom hardware cores and seamlessly integrated into the original software component architecture.

The underlying observation leading us to this approach is that, while software and hardware differ fundamentally at the algorithmic implementation level (as detailed earlier), the macroscopic mechanisms of data communication and synchronization are similar for both.

With this degree of hardware support provided, custom processing logic will be able to use
Figure 1.2: Distributed systems with accelerators: Classic master-slave configuration with distributed communication between host PCs (solid dark arrows) and local PC-accelerator communication (dashed dark arrows); our suggested semantic extension to direct accelerator-accelerator communication (light grey arrows)

the hardware API by using the same set of configuration parameters to control synchronization and data communication. Conventionally, FPGA components are utilized as co-processors or accelerators in a master-slave configuration as depicted in Figure 1.2. In this model, communication in a distributed system only happens between host PCs (dark continuous arrows); these PCs involve accelerators for their local workloads (dark dotted arrows). This forces programmers to treat and implement accelerated sections of the algorithm very differently, leading to discontinuities in programming style. In contrast, most traditional parallel programming models assume homogeneous processing elements even when assigning them different sub-problems. In our proposed model, hardware and software components can communicate with each other using the same protocols, and hardware components are treated as equal peers to software processes. Therefore “accelerators” (we prefer the less hierarchical term Reconfigurable computing hardware) do not just actively send data back to their local host and remote hosts, but also
directly to other “accelerators” (light grey arrows in Figure 1.2). For distributed systems in which the computation task is partitioned into a software-executed and a hardware-executed part, this partition can now extend beyond single systems. Host PCs and reconfigurable computing hardware comprise two different “computation planes” that can communicate and interact independently on their assigned tasks. As a further extrapolation of this independence, different configurations than the classical one-PC-one-accelerator structure are feasible, as for example a bank of independently communicating reconfigurable hardware platforms controlled by a single PC, or even no PC.

Previous work has successfully demonstrated this approach using the Message Passing Interface [18], which uses a distributed memory model. Message-passing is the most widely used programming model in high-performance computing because it is the closest approximation of the physical reality of most distributed systems.

Our presented work implements the same approach for the Partitioned Global Address Space (PGAS) model [19][20], which is a hybrid of the shared and distributed memory models. PGAS is a comparatively new programming model that is currently being widely explored in the High-Performance Computing research community. Its perceived advantage over message-passing is in productivity rather than performance: It provides a data access and communication model that is more intuitive than established distributed programming models, which enables the writing of clearer and more maintainable code. Furthermore, it enables minimization of communication costs by providing information about data locality to the programmer. Based on these attributes, PGAS has been gaining popularity and there are continuing efforts to develop languages, libraries and supporting software stacks to efficiently and productively use this model.

As we will discuss further, we consider PGAS a good model for the non-uniform memory architectures found in heterogeneous systems because its explicit, asynchronous memory transfers facilitate distributed programming with awareness of communication costs. PGAS is introduced in more detail in Section 2.


1.3 Research goals

Our work, as documented here, is part of an effort to provide a complete development flow, rooted in software design, that facilitates integration of reconfigurable computing hardware into software applications that can scale across more than one node. The previously mentioned advances in High-Level Synthesis and modern hardware description languages will increase productivity when designing customized hardware accelerators; they also make hardware design more accessible to a growing group of professionals and scientists that acquire software programming skills as an integral part of their education. However, they are only one component of what should be a comprehensive and coherent tool flow to develop heterogeneous systems. The existing High-Level Synthesis tools do not offer a clear path to isolate performance-critical functionality and provide a seamless path for migration to hardware without re-designing large parts of the existing software’s architecture. With a diverse range of disciplines accepting the necessity of software expertise and the resulting spread of coding education, any integration of specialized hardware should also be conceived starting from a software design perspective.

As a contribution to this approach, we investigate the integration of reconfigurable computing hardware into the PGAS distributed programming model. We are interested in whether FPGA integration can be facilitated in the different levels of a PGAS software stack in an organic way, i.e. to make reconfigurable hardware a natural component of the software architecture instead of a foreign body that disrupts the normal software design flow. Complementing this is the question of whether PGAS is a good model for the programming and integration of FPGAs: Are the computation, communication and storage properties of FPGA components reflected well in a PGAS model or does conforming to the model, on the contrary, possibly complicate efforts to build well-performing, heterogeneous systems?

To answer these questions in a satisfactory manner, we envision the following research efforts:

- To develop a complete framework for heterogeneous PGAS programming, composed of compatible software and hardware components according to the *Unified Programming Model* approach described in the previous section.
• To build and provide a heterogeneous distributed system (cluster) with independent CPU and FPGA nodes with the express purpose of implementing and evaluating PGAS applications based on the framework.

• To demonstrate and evaluate this platform for a range of applications with varying characteristics by implementing heterogeneous distributed solutions using the PGAS model.

• To comprehensively examine how PGAS and FPGAs fit together and what further steps need to be taken to establish a universal, heterogeneous PGAS solution.

In service to this vision, our work makes the following specific contributions documented in the subsequent chapters:

• We are introducing the first set of FPGA components that facilitate Direct Remote Memory Access, the direct memory-to-memory communication of data across networks of FPGAs and PCs. Our components abstract away specific networking and communication platforms and relieve designers of distributed hardware applications from the responsibility to implement scalable communication and synchronization mechanisms from scratch for each application. The control interface of our hardware components has been specifically designed to mirror GASNet [21], a widely used software API for remote communications.

• A software remote memory access library, based on the GASNet Core API, that has been specifically adapted to communicate across different processor platforms as well as with the FPGA components introduced above. By keeping the interfaces for the software library and the hardware components compatible, we offer a heterogeneous infrastructure in which software and hardware components can be exchanged against each other without changes to their external communication.

• By implementing three sample distributed computing applications with our infrastructure, we demonstrate a software-centric development flow that results in easier migration
of performance-critical components to FPGA hardware. We illustrate that the PGAS
programming model can be leveraged to represent heterogeneous and FPGA memory
infrastructure and we identify requirements to successfully implement heterogeneous dis-
tributed systems with the PGAS programming model.

1.4 Thesis organization

This thesis is organized in the following way: Chapter 2 discusses fundamentals of distributed
parallel programming and specifically the Partitioned Global Address Space model; it also
reviews related work. We introduce our heterogeneous framework and its software implement-
ation in detail in Chapter 3 while Chapter 4 explains the framework’s hardware components.
Chapter 5 presents our evaluation platforms and characterizes their basic performance num-
bbers through synthetic benchmarks. Chapter 6 introduces three sample application cases, their
software implementation and details their migration to hardware. Based on our results and
insights, Chapter 7 discusses the suitability of the PGAS programming model for distributed
systems that include FPGAs. Chapter 8 examines requirements to optimally integrate FPGAs
into PGAS frameworks. Chapter 9 concludes and discusses directions for future work.
Chapter 2

Background

In this chapter, we will briefly introduce common concepts of local and distributed parallel programming and examine relevant work in the areas of Partitioned Global Address Space programming and FPGA-aware programming models.

2.1 Forms of parallelism

To accurately situate our research, it is beneficial to differentiate briefly between the most common types of parallelism and how they can be exploited:

- **Instruction-level parallelism** is the parallelism inherent in non-dependent steps of an algorithm or a segment of code, e.g. two separate arithmetic operations necessary to provide inputs for a third operation. Hardware generally exploits instruction-level parallelism statically through pipelining and dynamically through re-ordering and superscalar execution units. Optimizing compilers can statically reorder instructions to provide the most opportunity for exploiting instruction-level parallelism.

- **Data-level parallelism** is resulting from the identical operation or string of operations being executed independently on separate data elements. A simple example is the addition of vectors or matrices of the same size. Vector processors (e.g. [22]) and vector instruction set extensions are hardware implementations that take advantage of data-level parallelism.

- **Thread-level parallelism** is a result of the availability of multiple streams of instructions.
These instructions streams could be exploiting data-parallelism by executing similar instructions on separate data elements or represent completely different actions being executed concurrently. The latter case can also be described as *task parallelism*.

Solutions for exploiting instruction- and data-level parallelism have existed in processor architectures since the 1960s. In the last two decades, thread-level parallelism has been exploited through *symmetric multi-threading* (SMT) in single cores, integration of multiple cores on single chips and *symmetric multi-processing* (SMP) systems with several processor chips. More recent SMP systems are employing *Non-Uniform Memory Architectures* (NUMA), introducing a locality component to main memory access cost.

In recent years, *accelerators* integrating large numbers of homogeneous simple processors have provided massive data-parallel processing capabilities to most computers. GPGPUs are common in most desktop computers (and increasingly, mobile devices) because of their original purpose of video output processing. Newer versions of NVIDIA’s *Compute Unified Device Architecture* (CUDA) support mapping of the GPU’s main memory into virtual address space; however, the memory’s physical separation through the PCIe bus still results in NUMA characteristics. GPU programming models also differ from CPU programming through more explicit management of the GPUs memory hierarchy.

FPGAs constitute a different type of accelerator, as implementations are statically exploiting instruction-level parallelism when they pipeline operations and instantiate concurrent data paths for independent parts of a computation. Designers can exploit data parallelism by instantiating multiple identical processing units. The dedicated main memories of FPGA devices as well as custom configurations of on-chip memories represent another manifestation of non-uniform memory.

This presence of non-uniform memory architectures even in desktop computers suggests that implementing algorithms using a flat memory model will lead to suboptimal performance. A programming model reflecting these changes needs to create awareness of the varying cost of memory accesses.
2.2 Distributed parallel programming models

Designers and programmers of High Performance Computing systems have considered the cost of data accesses, communication and synchronization for much longer than desktop-scale programmers. HPC systems are commonly designed as clusters of individual computing nodes connected by networks that are optimized for low latency and high bandwidth; network topologies provide a higher degree of connectivity than common local area networks. These network properties reflect an effort to minimize the effect of the physical separation of processing nodes from data located in the memory of other processing nodes. On the hardware-level, single nodes can generally just access their own memory through direct addressing. Separate from this physical reality is the question of what model of the distributed system to present to the application programmer. Figure 2.1 shows the three major variations established today.

2.2.1 Shared memory model

Shared memory programming models (left side of Figure 2.1) present the programmer with the opportunity to write code in a similar fashion as for a single computer. All data in the application is assumed to be accessible directly from the application code. Because of this property, another term used for this approach is Global Address Space. Global address space
programming on a distributed system implies that compiler and runtime libraries need to bridge the programmer’s logical view with the physical reality. Anytime a thread or a process accesses memory on another machine, the runtime environment needs to facilitate the data transport across the network invisibly to the programmer. It is easily apparent that inept application programming can lead to suboptimal results by causing expensive communication overhead. As a side effect of the single-machine view of the system, complete memory consistency as in a symmetric multi-processor system is assumed: This means that if an operation works on local data in each thread, and the subsequent operation works on remote data, barriers and memory fences need to implicitly assure this consistency across the system. This can lead to unnecessary synchronization overhead, as it is difficult for the compiler to differentiate between possible coherency issues and benign accesses or false sharing.

In summary, shared memory programming is highly convenient from the programmers perspective, as no explicit accounting for distributed systems is necessary, but it can lead to suboptimal performance because tuning the application for low communication and synchronization cost is adding considerable complexity.

A popular shared memory programming tool is OpenMP [24]. It uses compiler pragmas to split execution up into threads, to designate data and operations as parallelizable and to identify code segments that should only run on a single master thread. Basic OpenMP is only intended for symmetric multi-processing machines and therefore can lead to quite effective code at low programming cost. Its distributed extension Cluster OpenMP [25] has found limited adoption because of the discussed issues.

2.2.2 Distributed memory model

A distributed memory programming model (right side of Figure 2.1) gives the programmer explicit awareness of the separate physical memories in a cluster. Application threads or processes can only directly access their local address space. Access to remote data can only be accomplished through explicit network communication; the character of this communication is inherently two-sided, as the programmer needs to setup send and receive operations in the two communicating processes. As a consequence, it is more difficult to efficiently overlap computation and communication. The explicit nature of communication leads to the model’s alternative
name of Message-passing.

The Message Passing Interface (MPI) [26] standard is a widely adopted API specification for message-passing. Though numbers are hard to come by, it is commonly accepted that the large majority of all distributed computing applications either use MPI directly or use a domain-specific library based on MPI. This is because the programming model’s explicit communication mechanisms offer the best opportunity to maximize application performance. The trade-off is more cumbersome programming as all remote data accesses involve more complicated operations.

### 2.2.3 The Partitioned Global Address Space model

The more recently introduced Partitioned Global Address Space model or PGAS (centered in Figure 2.1) is an attempt to combine the beneficial properties of shared and distributed models. PGAS languages and libraries enable direct access to data on all nodes in the system, but they allow programmers to designate and handle data with locality properties, e.g. private vs. shared or local vs. remote. The explicit information about a data element’s locality allows programmers to minimize expensive memory operations. At the same time, remote data accesses can still be handled implicitly, allowing a more elegant and intuitive programming style.

Central to the idea of PGAS is the concept of one-side accesses: Semantically, an application process can read and write remote data without involving the application process that the data belongs to (the actual access at the target machine can be implemented through hardware support or dedicated threads or processes). This is in contrast to the two-sided approach used by message-passing models, where both the sender and the receiver need to set up any data transfer. This has a large impact on how applications with irregular remote access patterns can be implemented.

Distributed Shared Memory would be a good term to describe PGAS, but it has already been taken by earlier Global Address Space approaches [27][28][29] that hew closer to the properties and drawbacks of what we classified here as Shared Memory.

Partitioned Global Address Space is not a very narrowly defined model, and it does not have one standard library or language extension. A host of language variations, libraries and newly designed languages are available and in continuous development. We are presenting the
most common implementations as part of the Related Work section that follows.

2.3 Related Work

2.3.1 PGAS implementations

Modified languages: Unified Parallel C, CoArray Fortran, Titanium

*Unified Parallel C* (UPC) [30] is a conservative extension of standard C, with the goal of making PGAS concepts available with minimal performance overhead. A fundamental change is the ability to designate data as *private* or *shared*. If a variable or data structure is (implicitly) declared private, only the local thread can access it. If a scalar variable is designated as shared, all threads access the variable located in thread 0. If an array is declared shared, it is distributed across all threads in a specified manner (block, cyclic, block-cyclic). Multi-dimensional arrays are still implemented as arrays of pointers like they are in C. UPC offers dynamic memory allocation across threads and has a shared pointer model to reference remote data. Other features include barriers, locks, the ability to specify memory consistency per variable, code block or program as well as a modified *for* statement that allows selective execution on thread-local data.

*Co-Array Fortran* [31] and *Titanium* [32] are PGAS extensions to the Fortran and Java languages in similar ways as UPC is to C. The recently introduced *UPC++* [33] applies PGAS concepts to objected-oriented programming with C++.

Application libraries: OpenSHMEM, Global Arrays

*OpenSHMEM* [34] is a popular library providing one-sided remote memory transfers, collective and synchronization operations to application programmers. It is similar to the GASNet Extended API, which we will introduce and examine in Section 3.3. The SHMEM (Symmetric Hierarchical MEMory access) library was originally introduced by Cray and is considered the first one-sided communications solution for distributed systems. Several almost compatible implementations by other HPC and network vendors were subsequently introduced. OpenSHMEM is an open source effort to create a modern reference implementation.
Global Arrays [35] is an early implementation of PGAS principles that is still widely used. It enables working with distributed multi-dimensional arrays and provides functionality for allocation, distribution schemes, local copies of array sections and synchronization. Global Arrays can be linked against C, C++, Fortran and Python. Global Arrays uses its own one-sided communication library called ARMCl [36], which is similar to SHMEM and GASNet.

New PGAS languages: Chapel, X10

Cray and IBM have developed new objected-oriented PGAS languages from the ground up with Chapel [37] and X10 [38]. Some of their most prominent features are:

- **Global view execution model**: In contrast to an SPMD Single-Program-Multiple-Data execution model as used for example by MPI and UPC, a single program is semantically understood to be executed on the whole distributed system, with execution of parts of the algorithm assigned implicitly through the locality of data or explicitly by starting tasks on specified nodes. “Private” scalar values identical over all nodes will be calculated locally everywhere, minimizing communication for certain serial code segments.

- **Native multi-dimensional arrays** and tools to specify their distributions and overlapping regions of interest (e.g. for stencil operations).

- **Locales/Places**: Both languages can identify subcomponents and groups of components in an overall distributed system that share local properties like one physically accessible address space, e.g. a single SMP system in a cluster. These locales or groups of locales can be assigned ownership of specific data and specific tasks can be explicitly spawned on them.

- **Implicit asynchronicity**: Both languages make explicit synchronization a language feature, e.g. X10 with async/finish and Chapel with coforall/cobegin/sync.

- **Immutables**: Data objects that, in contrast to variables, cannot be modified after being initialized with a value. Similar to compile-time constants, immutables can give compilers more opportunities for optimization.
2.3.2 Previous work on PGAS and FPGAs

On the PGAS side, SHMEM+ [39] is an extended version of the established SHMEM library that uses the concept of multilevel PGAS as defined by its authors: Every processing node has multiple levels of main memory, e.g. CPU main memory as well as an FPGA accelerator’s main memory, which are accessed differently by SHMEM+. For node-to-node transfers, GASNet is used, for transfers to a local or remote FPGA, a vendor-specific interface has to be accessed by the local CPU.

El-Ghazawi et al. [30] examine two different approaches to use FPGAs with Unified Parallel C: In the library approach, a core library of FPGA bitstreams for specific functionalities exists. Function cores can be explicitly loaded into the FPGA. An asynchronous function call transfers data for processing into the FPGA, a later completion wait transfers the processed data back into CPU-accessible memory. The second approach uses a C-to-RTL synthesis of selected portions of the UPC code: A parser identifies \texttt{upc\_forall}-statements that can be well parallelized in hardware and splits their compilation off to Impulse-C [40]; corresponding data transfers to and from the FPGA are inserted into the CPU code.

Our common concern with the two PGAS solutions is that they leave the CPU(s) in charge of all communication management, while the FPGA remains in a classic, passive accelerator role. Truly efficient one-sided communication as embraced by PGAS is therefore not available hardware-to-hardware, and FPGA capabilities might be underutilized.

Finally, RAMP Blue [41] is a massive multiprocessor platform, with the intent of supporting multiprocessor research through use of soft processors on FPGAs. Like our work, it also uses MicroBlaze processors connected through a FIFO-based network scalable across multiple FPGAs and FPGA boards. It uses the NAS Parallel Benchmarks [42] based on UPC for performance testing. To enable this, Berkeley UPC and the underlying GASNet were ported to the MicroBlaze platform and the FIFO-based conduits. While the successful porting of the full GASNet library is an impressive feat, there is no intent to extend the software implementation to hardware accelerators; the MicroBlaze itself is a stand-in for future multiprocessor cores. In contrast, our work focuses on a low-footprint GASNet Core library for the MicroBlaze that can communicate across platforms; the intent is that the MicroBlaze serves as an intermediate
stage of the application design process and is later replaced by hardware accelerators that can communicate in the same way.

### 2.3.3 Other related work

The approach of having a similar, migration-conducive API for software and hardware components has been successfully used by Saldana et al. [18]. TMD-MPI is a library that implements a subset of the MPI standard to enable message passing between FPGA and CPU components. It has successfully demonstrated a common communication API for software and hardware components for the application area of molecular dynamics simulation. Our work is inspired by and still partially based on TMD-MPI infrastructure. TMD-MPI’s downsides are the ones incumbent to any message-passing system: The need for low-level transfer management, two-sided communication and its scalability limits, the inefficiencies of indirect communication to remote memories and the ensuing impediments to work on dynamic data structures distributed across nodes.

Hthreads [43] is a hybrid shared-memory programming model that focuses on implementing FPGA hardware in the form of real-time operating system threads, with most of the properties of software, but adding real concurrency. Hthreads focuses on components sharing buses with each other in single-chip processor systems, and lacks a scalable programming model for multi-node systems.

LEAP (Logic Based Environment for Application Programming) [44] is a Virtual Platform Architecture that offers hardware designs an operating system-like set of abstractions and services including main memory allocation, caches and communication channels with user-defined protocols between FPGA components and software. Its abstraction layer for FPGA applications has been implemented for several platforms with both Xilinx as well as Altera devices. There are many similarities to our work in the platform abstractions, however LEAP is not focused on large-scale parallel systems and scalability.

CoRAM [45] is an on-FPGA infrastructure to abstract external memory interfaces from hardware processing cores. CoRAMs are instances of on-chip memory that can be directly accessed by the processing unit on one port. A second port is connected to finite state machines that execute so-called control threads, which manage transfer of data to and from off-chip RAM.
The processing core is therefore completely abstracted from the external memory interface. The use of different control threads enables the on-chip RAM to function as scratchpad memories, caches or FIFOs. CoRAM does not focus on communication with remote memories, and does not take distributed programming models into consideration. It is principally intended to provide an abstraction to external memory. However, the concept of control threads is similar to one of our components (see Section 4.2.3). CoRAM++ [46] adds support for data structure-specific access management for multidimensional arrays, linked lists in addition to block and stream access.
Chapter 3

THeGASNet - A heterogeneous GASNet implementation

In this chapter, we will describe a software communications layer for PGAS systems, called THeGASNet, that we have implemented with heterogeneous systems in mind. It is an extended implementation of a pre-existing software API, GASNet, that is used by multiple PGAS languages and libraries. Chapter 4 will then subsequently describe how we mirrored the software functionality described here in hardware.

3.1 Structure of a generic PGAS stack

Figure 3.1 illustrates a generic PGAS software stack: At the bottom is the network infrastructure, which here encompasses networking hardware, drivers and any communication protocols utilized by the layers above. On top of this is a one-sided communication library where one-sided implies that it enables remote memory access without involvement of the remote application. By enabling access to memory declared as shared on all locations in a distributed system, it creates a Global Address Space (GAS); this term does not necessarily imply a contiguous virtual address space across the system. Based on the one-sided communication facility, a PGAS language or a PGAS library enables the application on top to manipulate data objects everywhere in the global address space. The Partitioned in PGAS means that the language or library provides a way for the application to distinguish between local and remote data, enabling it to
optimize communication costs.

Our approach, as presented in Section 1.2, involves using two very different forms of implementation at the application layer, software code to run on CPUs and custom accelerator hardware to work in the FPGA fabric. However, we want to make them work side by side in the same system. They have in common that they access data in local memory and process it, however possibly in quite different ways, and they are both reliant on lower levels of the stack to access remote data. As our intended design process involves making software and hardware exchangeable at the application level, we need them to access one of the lower levels with a common interface.

At the other end of the stack, network infrastructure differs between CPU-centric systems and FPGAs, too. CPUs will access a network stack involving software protocols and drivers and commodity network hardware. FPGAs will have on-chip networks and off-chip networking interfaces, often custom solutions, but no software layer in-between. The layer that offers the same interface to both implementation variations also needs to abstract away those differing network infrastructures. Among the two remaining stack layers in the middle, we choose the one-sided communication layer to present a common interface to both types of applications. Remote memory access needs to be used for both application varieties: Communication and synchronization of data is required by both software and hardware implementations, and we
would like them to behave similarly or identically to facilitate migration from one to the other.

However, PGAS language runtime or library layers are likely to look quite different for the two varieties. Software applications are written in high-level languages like (or similar to) C/C++, Java or Fortran. Hardware applications are either written in hardware description languages or converted to hardware description languages by High-Level Synthesis. The long term vision is to be able to use identical software code for both CPU code and as source for hardware generation; however, the constraints and limitations imposed by current High-Level Synthesis tools rarely facilitate this yet. Even in this case, while the algorithm has been translated from high-level language to HDL, the resulting interfaces are still more “HDL-like”. We will first test with application use cases how a similar one-sided communication layer interface works with software as well as hardware applications, and then draw conclusions from these experiences about the best nature of PGAS language or library runtime layers for both variants.

3.2 Choosing a communications API for heterogeneous systems

Having decided that a common remote communication layer is our intersection point for software and hardware components, we need to define a specific interface and features for this layer. In common software terms, we are talking about an Application Programming Interface or API. Among the feasible choices for such a communications API are:

- Defining a completely new API suited to our needs
- The OpenCL [47] API used to access General Purpose GPUs
- The distributed-memory Message Passing Interface (MPI) [48]
- The Global Address Space libraries OpenSHMEM [34], ARMCI [36] and GASNet [49]

Starting with an API definition from scratch obviously holds the appeal of being able to tailor all its properties exactly to our requirements and goals for a unified API. However, we consider a custom API for heterogeneous distributed systems as a possible outcome, not a starting point, to our research. We are aware of our limited depth of experience at the outset of a project. An
established system that has proven itself to a larger user community and has been refined over multiple design iterations is likely to include practical considerations we might overlook in the beginning and which we will only start to appreciate while accumulating practical experience. Having acquired this experience, it is much easier to judge the strengths and weaknesses of an existing platform and lay out improvements for a future solution.

OpenCL is an interesting option as it is already established as a heterogeneous programming interface to utilize GPGPUs (its initiators advertised OpenCL as a general interface suitable to access different kinds of accelerators including DSPs, but the similarities to NVidia’s CUDA framework make its GPU focus hard to deny). Its exposition of memory hierarchy and explicit, sometimes asynchronous, data transfers arguably make it a PGAS variation. As a consequence of OpenCL’s popularity, both major FPGA vendors have put forward solutions combining High-Level Synthesis with an OpenCL interface to use their FPGAs. We observe these approaches with much interest and acknowledge that they can solve both the FPGA interfacing and programming issues for a subset of existing problems and deliver FPGA acceleration with limited design effort. In our opinion, they lack universality, as they overconstrain FPGA capabilities to a massively data-parallel GPU structure, although without the GPU’s high clock speeds and large amounts of on-chip memory. Furthermore, they currently offer no scalability beyond a single host-accelerator node.

A unified MPI solution for software and hardware has been successfully implemented and used in molecular dynamics computations by previous research [18]. As we consider the PGAS characteristics of asynchronous, one-sided memory accesses as a useful property, MPI in its current form is not a good choice for us. More obvious candidates are therefore the OpenSHMEM [34], ARMCI [36] and GASNet [49] APIs, as they are all part of software stacks used by PGAS applications. All three libraries offer high-level functions for unlimited-size remote memory reads and writes as well as synchronization primitives. OpenSHMEM offers several types of collective operations. ARMCI supports strided and vectored(scatter/gather) remote operations; these operations have also been suggested for GASNet, but so far have not been added to its specification.

We have decided to use GASNet because its specification is the only one to expose a lower API layer, the Core API, which is used to implement the more complex functions at its top
level, the Extended API. The Core API, which is described in more detail in the following section, exposes communication with hardware-limited message sizes and allows messages to trigger specific functionality on arrival. This level of functionality is in our opinion most suitable to specify heterogeneous communication, since higher-level functionality might be implemented differently on FPGA or CPU platforms\(^1\).

In this chapter, we introduce Toronto Heterogeneous GASNet (THeGASNet), a heterogeneous implementation of GASNet that allows portable, fully compatible communication across FPGA hardware as well as several processor platforms. To be able to differentiate our design modifications, we will first introduce GASNet in detail.

### 3.3 The GASNet API and library

GASNet (Global Address Space Networking) is a remote memory communication library that is used by UPC, UPC++, Co-Array Fortran, Titanium and Chapel. It provides the functionality to send messages or blocks of data between nodes in a distributed system, trigger actions on remote nodes and synchronize execution between nodes. In contrast to MPI, GASNet enables direct writes to as well as reads from another node’s shared memory without explicit involvement of that node’s application thread(s). It therefore enables one-sided communication, leading to

\(^1\)Subsequent work by Pandit et al. [50] explores a potential approach to implement the Extended API for heterogeneous platforms on top of our Core API.
a potential performance advantage of PGAS implementations over message passing.

GASNet is divided into two major layers, which can be seen in Figure 3.2. The GASNet Core API defines so-called Active Messages, which can copy data packets of limited size into another node’s shared memory. GASNet is adapted to a specific network infrastructure by implementing the Core API using the provided network functionality. In the GASNet nomenclature such a network-specific implementation of the Core API is called a conduit [21]. The Core API also includes all functions necessary for setting up basic GASNet operations on a node and link up all nodes into a distributed system.

The GASNet Extended API offers blocking and non-blocking Put and Get data transfers, whose size is only limited by the host architecture, as well as various barrier functions. A complete implementation of the Extended API only using Core API calls is provided as part of the GASNet source code, although some higher-capability network interfaces might provide functionality to directly implement some Extended API calls, too.

In common usage, the GASNet APIs will only be called by the runtime library of a PGAS language, e.g. the Unified Parallel C runtime. However, in this work we will demonstrate distributed applications using the API directly, as we currently lack a suitable PGAS language or library targeted to the heterogeneous systems at the center of our research. The complexity level of using GASNet directly is comparable with the one using MPI, with the distinction of providing one-sided memory transfers².

### 3.3.1 Active Messages and Handler functions

The GASNet Core API’s remote memory operations are implemented by Active Messages, a concept first introduced by von Eicken et al. in [51]. The Core API defines three major types of Active Messages:

- **Short** messages do not carry a data payload read from or written to memory. They are intended to carry small amounts of information that are specified as function arguments from node to node. As such they can also be used as a signaling mechanism.

---

²In its recent revisions, the MPI standard added Remote Memory Access (RMA) capability, however implementation support and adoption are still limited.
• *Long* messages are used to transport data directly from a local node’s memory to a remote node’s designated shared memory.

• *Medium* messages are used to transport data from a local node’s memory to a temporary buffer on the remote node. Handling of the received message must therefore include processing or copying this data in some way, as the buffer will be de-allocated when the transaction is complete. Medium messages can be especially useful for data transfers that take place before the remote machine’s shared memory segment address and size have been published.

A function call to generate an Active Message takes the following parameters:

• The *destination node* for the message

• A *handler function* ID. This number specifies a pre-declared handler function that will be called on the remote node as soon as the data has been copied into the target memory location (or directly after the message has arrived, in case of the *Short* type).

• A variable number of 32-bit function arguments which will be used as parameters in the handler function call. The maximum number of arguments is platform-specific, but no less than necessary to specify eight memory pointers on the platform.

• A pointer to the local source memory (in case of *Medium* and *Long* messages)

• The size of the block to be transferred (in case of *Medium* and *Long* messages)

• A pointer to the destination address on the remote node (in case of *Long* messages)

GASNet’s one-sided access semantics imply that applications are not directly involved in the reception and processing of Active Messages. Instead, an Active Message triggers a *Handler function* on the receiving node that is concurrent to the application thread. Implementations of the handler mechanism are platform-specific and can consist of polling- or interrupt-based mechanisms as well as multi-threading.

Handler functions are called *after* a possible data payload has been copied into the remote node’s memory. They can process the received data in some way, signal arrival of data to the
application or initiate a reply to the initial message. To preclude deadlocking, Active Messages have strict Request-Reply semantics: The handler of an initial Request Active Message can only send one Reply Active Message back to the initiating node.

A handler function takes the following parameters when called upon message reception:

- A sender token. To reflect the request-reply semantics, handler functions cannot directly address another node in a reply, but only specify the token. GASNet will then route the reply message back to the original sender based on the token.

- A (statically specified) number of arguments that this particular handler function can accept.

- The size of the received data block (in case of Medium and Long messages).

- The pointer to the temporary data buffer (in case of Medium message) or the final data destination (in case of Long messages).

A remote write using a single Active Message is visualized in the two parts of Figure 3.3. As can be seen in part (A) of the figure, the application on the left node calls the GASNet API, requesting a Long type active message (1), specifying the payload data in its virtual address space and its destination address in the remote node’s designated shared memory area. GASNet (2) generates a network packet consisting of the ID of the requested handler function, the arguments for the handler function and the payload data and sends it off through the network (3). Part (B) of the figure illustrates that on the receiving node, GASNet arranges for the payload data to be written to the specified address in the shared area (4). When the write has concluded, it calls the specified handler function with the supplied arguments (5).

Figure 3.4 depicts a remote read, a common application for the request-reply pattern: The initiating application process calls GASNet (1), specifying a remote node, a shared memory address on that node, a data size and a local memory address to read the data into. All these parameters are transmitted as handler arguments in a Short type message (2). On the remote node, GASNet calls up a handler function responsible for remote reads (3) based on the message’s handler ID. The handler function uses the provided parameters to initiate a Long type reply (4). With the specified shared memory segment as payload (5), GASNet sends off
Figure 3.3: Remote write
an Active Message to the initiator (6). On the initiating node, the received payload is written to shared memory (7) and a handler function is called to register the transaction’s completion. The whole process is inherently asynchronous as neither node’s application process needs to stop execution flow and wait for it to complete.

### 3.3.2 Sequence of GASNet node operations

The basic processing unit in GASNet is the node. A node is defined as a single processing unit that runs an instance of the application in directly addressable local memory and that has a unique node identifier in the whole distributed system. Depending on the application, library or language runtime using GASNet, it can be represented by a single processor, a single operating system process or a single thread (which might spawn further threads for thread-level and data parallelism) accessing an instance of the GASNet API. In our heterogeneous concept, it can also be an FPGA custom hardware block controlling its own GASNet hardware core.

A software application’s `main()` function has to go through several GASNet-specified initialization steps to connect to the whole system as a unique node and to enable full GASNet operations:

1. Call `gasnet_init()` with references to the standard C `argc` and `argv` arguments. This func-
tion initializes GASNet operations on any node before any processing or communication take place. The function \texttt{gasnet\_init()} can extract GASNet-specific commandline arguments from the argument vector and modify its size and contents. It therefore has to be called before any argument-dependent application operations. After this function call, an application can inquire about basic system properties like the number of GASNet nodes and its own node ID, but not yet initiate or receive Active Messages.

2. Call \texttt{gasnet\_attach()} with a \textit{handler table} and the requested size for a shared memory segment. The \texttt{gasnet\_attach()} function sets up node-to-node communication and allocates a shared memory segment for the local node. The first parameter to it is an array of \texttt{gasnet\_handlerentry\_t} structs, which holds the function pointers and handler IDs for all handler functions that the application has defined. These entries are combined by \texttt{gasnet\_attach()} with the handler functions that GASNet uses internally to maintain operations (different handler ID ranges are defined by the GASNet standard for internal and application use respectively). After allocating the shared memory segment and storing its size and address, the function activates any interfaces necessary for communication with other nodes. After the function completes, the application can send \textit{Short} and \textit{Medium} messages. Long messages are still impossible as the application does not yet have information about shared memory on other nodes. Similarly, GASNet can at this point start processing messages from other nodes and initiate the appropriate handler functions.

3. To be able to address shared memory - both its own as well as segments on other nodes - the application needs to call \texttt{gasnet\_getSegmentInfo()}. This function provides a table that holds the virtual start address as well as the size of the shared memory of each node in the system. Incidentally, this is the only way that an application can find out its own shared segment address, as this information has up this point only been held internally by the node’s GASNet instance. With successful completion of this function, a node is able to send \textit{Long} Active Messages.

As \texttt{gasnet\_getSegmentInfo()} needs to exchange information between all nodes, it is a \textit{collective} function, in contrast to the \texttt{gasnet\_init()} and \texttt{gasnet\_attach()} functions.

GASNet also provides a function \texttt{gasnet\_exit()} to terminate a finished application on the local
node. However, this is not a collective operation, so it should only be called after synchronizing application finish with a global barrier.

### 3.3.3 GASNet Core API: code example

The following code example demonstrates use of the GASNet Core API for a simple application involving remote memory copy. A text string is copied from node to node and printed out at each receiving node, finally arriving again at the initiating node. Detailed explanations follow behind the code segment. To keep the code simple, function return checks are not included.

```c
#include <stdio.h>
#include <string.h>
#include "platforms.h"
#include <gasnet_core.h>

// function prototype for barrier
void barrier();

// GASNet handler function prototypes
#define hID_S0_barrier_increment 140
void S0_barrier_increment(gasnet_token_t token);

#define hID_L0_receivestring 131
void L0_receivestring(gasnet_token_t token, void *buf, size_t nbytes);

// handler function table
static gasnet_handlerentry_t handlers[] =
{  
    {  
        hID_S0_barrier_increment,  
        (void(*)())S0_barrier_increment  
    },  
    {  
        hID_L0_receivestring,  
        (void(*)())L0_receivestring  
    }
};

// global variables
static unsigned int barrier_cnt = 0;
static unsigned int received = 0;
static char *rcvstring;
```
int main(int argc, char **argv) {
    gasnet_seginfo_t *segment_table_app;
    void* mysharedmem;
    size_t sharedmem_size = 0x100000; // 1MBbyte shared mem to allocate
    unsigned int nextnode;
    void* nextnode_sharedmem;
    ptrdiff_t string_offset = 1024; // position in shared memory
    char *teststring = "Mr.Watson, come here, I want to see you."

    // gasnet start-up
    gasnet_init(&argc, &argv);
    gasnet_attach(handlers, sizeof(handlers)/sizeof(gasnet_handlerentry_t),
                  (uintptr_t)sharedmem_size, (uintptr_t)0);
    segment_table_app = malloc(gasnet_nodes() * sizeof(gasnet_seginfo_t));
    gasnet_getSegmentInfo(segment_table_app, gasnet_nodes());

    // find nextnode, shared memory segments
    mysharedmem = (void*)segment_table_app[gasnet_mynode()].addr;
    nextnode = (gasnet_mynode() + 1) % gasnet_nodes();
    nextnodesharedmem = segment_table_app[nextnode].addr;

    // GASNet is initialized
    printf("Node %d ready.
",gasnet_mynode());
    barrier();

    // First node: Send off string
    if (gasnet_mynode() == 0)
        gasnet_AMRequestLong0(nextnode, hID_L0_receivestring,
                               teststring, strlen(teststring)+1,
                               nextnode_sharedmem+string_offset);

    // All nodes: Waiting for message receive; could do other stuff meanwhile
    while(!received);

    // All other nodes: Send string on
    if (gasnet_mynode() != 0)
        gasnet_AMRequestLong0(nextnode, hID_L0_receivestring,
                               (void*)rcvstring, strlen(rcvstring)+1,
                               nextnode_sharedmem+string_offset);
```c
printf("Node %d received at address %p: %s\r\n",
gasnet_mynode(), rcvstring, rcvstring);

barrier();
printf("Node %d done.\r\n",gasnet_mynode());
barrier();
gasnet_exit(0);
return 0;
} // end of main()

void barrier()
{
    if (gasnet_mynode() == 0) {
        // wait for messages from all others
        while(barrier_cnt < gasnet_nodes()-1);
        barrier_cnt = 0; // reset

        // send complete messages to all others
        for(int t=1; t < gasnet_nodes(); t++)
            gasnet_AMRequestShort0(t, hID_S0_barrier_increment);
    } else {
        // Send message to 0
        gasnet_AMRequestShort0(0, hID_S0_barrier_increment);

        // wait for complete message from 0
        while(barrier_cnt == 0);
        barrier_cnt = 0; // reset
    }
}

// GASNet handler functions
void S0_barrier_increment(gasnet_token_t token) {
    barrier_cnt++;
}

void L0_receivestring(gasnet_token_t token, void *buf, size_t nbytes) {
    rcvstring = (char*)buf;
    received = 1;
```
The obligatory header includes at the top of the code are followed by function prototypes. The application defines one regular function, `barrier()`, that implements a synchronizing barrier between nodes with the help of Short-type Active Messages, as explained further below. There are two further prototypes for handler functions that will be called by GASNet on reception of Active Messages, `S0_barrier_increment()` and `L0_receivestring()`, which handle the Long-type messages copying the strings. Both prototypes are accompanied by handler ID definitions: An application can assign IDs from 128 to 255 to different handlers (0 to 127 are reserved for internal GASNet use). The prototypes are followed by an array that pairs the handler functions pointers with the corresponding IDs.

Finally, three global variables are defined for each node that will be used to communicate between the `main()` function and handler functions and to hold persistent values between handler calls.

The `main()` function starts by defining local application variables:

- A pointer to the segment table, a table that will hold the virtual addresses and sizes of all nodes’ shared memory segments
- A pointer to the local shared segment
- The requested size of the local shared segment
- The node ID of the next node, the destination for the string copy
- A pointer to the virtual address of the next node’s shared segment
- The address offset for the string destination in the next node’s shared segment
- The string to be copied

The call to `gasnet_init` in line 47 provides the GASNet library with command-line parameters for initial setup of appropriate data structures. GASNet communication is possible only after the subsequent `gasnet_attach` call. With the provided parameters it sets up an internal table to call up the correct handler functions based on ID, not name, allocates a local shared memory segment of the requested size, and sets up any communication channels.

Before a call to `gasnet_getSegmentInfo` (Line 53) can determine shared memory information about other nodes, the application needs to allocate a segment table for this data (Line 51).
This can only happen after the call to `gasnet_init` as only then the absolute number of nodes can be determined with a call to `gasnet_nodes()`.

With the segment table, pointers to the local shared memory and to the destination node for the string copy can be initialized with the correct information. Determining the next node ID includes a wraparound by modulo so that the last node points to the first, node 0.

Lines 62 and 63 start off application operation with a printed message and a (in this example, redundant) barrier to synchronize operation on all nodes.

In a pattern common for SPMD (Single-Program-Multiple-Data) applications, control flow diverges depending on the specific GASNet node ID. On the first node (numbered 0), a text string is locally allocated and remote-copied to the next node (numbered 1) by requesting a Long Active Message in line 67. Arguments to the function are the destination node, the handler ID, the address of the local data, the size of the local data, and the address on the destination node. Note that the local address does not need to be part of the shared memory segment: In our case, it is a string fixed in the local read-only segment at compile time. However, the remote address always has to be in the shared segment: It is specified as the start of the next node’s shared segment plus the pre-defined offset for the intended string position.

The next step in the code is executed by all nodes: A while loop waits for the global variable

---

Figure 3.5: Remote string copy example: Private node memory is grey, shared memory is white. LongAM denotes an Active Message of type Long (holding a data payload).
received to be changed from 0 to 1 and thereby signal the arrival of data (See line 72). While in this case, the loop case doesn’t do anything else, the asynchronous nature of the one-sided communication means that the application could do other work and only check on the condition when the received data is required.

When the data has been received, all nodes except the first node react by sending the data on to the next node (if node 0 receives the string, the sequence of remote copies is finished). They use the same Active Message request as node 0 did, except that the source address now is in the shared segment, specifically the destination of the previous transfer from the previous node.

After sending the string on, each node prints the received string in line 80. All nodes call another barrier after this to synchronize completion of the copying process before printing that they are done in line 84. Another barrier synchronizes after the completion printout before gasnet_exit() gets called on each node in line 87. The whole process is visualized in Figure 3.5.

The definition of the function barrier() begins in line 92. There is again a diverging control flow based on the node ID. A barrier is a collective operation that needs to be controlled by one node. In this case, node 0 will wait to receive barrier messages from all other nodes, which are counted through the global variable barrier_cnt. When all messages from other nodes have arrived, therefore fulfilling the synchronization condition, node 0 sends messages back to all other nodes to complete the operation. For the nodes other than 0, the order of actions is reversed. They send off one Short Active Message request to node 0 right after entering the function, and then wait for node 0’s Short Active Message indicating completion. Both flows react to their completed wait condition by resetting the variable barrier_cnt to 0 for the next barrier.

The barrier mechanism is dependent on the handler function S0_barrier_increment(), which is listed starting at line 114. It only executes one action, increasing the global variable barrier_cnt.

The last function in the code is the handler function L0_receive_string() at line 119. It is called after the remote string copy has finished and executes two actions: First, it writes the pointer to the new data into the global string pointer rcvstring so that it can be used for the next message request and the print output. Then, it sets the global variable received to 1,
3.4 Enabling cross-platform compatibility for THeGASNet

A major difference between GASNet and THeGASNet is our intention for API communication to be compatible between several CPU types and custom FPGA components. This section describes the API modifications required to facilitate this.

3.4.1 Target architectures

Figures 3.6 to 3.8 show the architectures we are targeting for use with THeGASNet:

- Processors with the x86-64 instruction set architecture that serve as CPUs in a desktop or server PC system (system (a) in Figure 3.6). Processors can run multi-threaded applications on top of the Linux operating system. The system can communicate with other nodes by IP over Gigabit Ethernet, and with in-system or external FPGA accelerators by PCIe.

- 32-bit multi-threaded embedded processors based on the ARMv7 architecture, running the Linux operating system. As part of a Xilinx Zynq platform (system (b) in Figure 3.6),
they connect to an FPGA fabric through the ARM AXI bus. The FPGA fabric can also directly access the external main memory shared with the ARM CPU. The ARM system on Zynq has a Gigabit Ethernet controller connected to the processor’s host bus.

- Xilinx MicroBlaze processors are 32-bit, single-threaded soft processors implemented in an FPGA’s programmable fabric (see Figure 3.7). Due to its small size, contemporary FPGAs can hold a significant number of MicroBlaze systems at the same time. External memory can be shared between multiple processors through an AXI bus. In our design flow, MicroBlaze processors will mostly function in two roles: As an intermediate development stage for software-to-hardware migration, or as a control node to manage hardware computation cores if no x86 or ARM host processor is available.

- Application-specific custom processing cores implemented in the hardware fabric (see Figure 3.8). Inherent to custom hardware, these cores can be of any required structure. To make custom cores interoperable in our design flow, we will however require them to use the same main memory and networking interfaces as the MicroBlaze processors, enabling substitution of the processors with custom engines.

\[\text{Massive multiprocessor systems on an FPGA are feasible (See RAMP Blue [41]); however, low operating frequencies and limited on-chip storage capacity make such systems uncompetitive with hardwired massive multiprocessor systems like GPGPUs and Intel MICs [52].}\]
3.4.2 Interoperability issues

To ensure smooth interoperability between the introduced platforms, the Active Messages that are the fundamental unit of GASNet communication need to use a standard representation in memory and throughout all network communication.

A common interoperability problem is Endianness, the order of representation of data words larger than the basic unit of a byte. Fortunately, all our current target processors use LittleEndian representation. Additionally, both the byte-oriented Internet Protocol and the Xillybus PCIe cores that we are employing naturally conform to this. Therefore, currently no accommodation of endianness is necessary, but awareness of this issue is important when adding further processor and communication platforms.

Another interoperability issue is the size of pointers, which represents the size of virtual and physical address spaces. As we have both a 64-bit processor and hardware than can address more than 4GBytes of memory among our target platforms, we are using 64-bit address information throughout our communication. Address information that stays strictly local, e.g. the copying of data inside a single processor’s address space, can keep using native pointers, which are only 32 bits wide in the ARMv7 and MicroBlaze architectures.

A further restriction involves the message size. We have decided to limit the current maximum size of an Active Message’s data payload to 8 KBytes. With added header information,
an Active Message will still fit into a single Ethernet *Jumbo frame*. Furthermore, the resulting message size can still be buffered at reasonable resource cost in FPGA on-chip buffers. As a consequence of this restriction, no size parameter inside a message header currently requires more than 14 bits, although up to 32 bits are allocated in most places for easier word-aligned data handling.

### 3.4.3 Packet format

To fulfill the requirements listed above, we are using a basic Active Message data format as visualized in Figure 3.9. Its base units are 32-bit/4-byte data words and it is used both for network packets and for in-memory storage of Active Messages. The first word of the packet specifies the GASNet source node, the destination node and the number of following 32-bit
words. Therefore, it holds all the information needed to route the complete packet. Currently, 8 bits each are allocated for the source and destination nodes, limiting the overall system size to 256 GASNet nodes\(^4\). Following the network header is a block of words describing all the relevant attributes of the Active Message; an example for the Long type message can be found in Table 3.1. A detailed overview of all THeGASNet communication formats is available in Appendix B. Following the message properties are the 32-bit arguments intended as parameters for the handler function. The final section of the packet is constituted by the data payload to be transferred by the message.

### 3.5 THeGASNet software library for regular CPUs

#### 3.5.1 Implementation strategy

To build a heterogeneous extension to an existing API, we have considered two approaches:

1. *Modifying and extending the existing Berkeley GASNet implementation*: The main arguments for extending the existing open source implementation would be to make our work available to an established community of users and taking full advantage of the well-tuned performance and scalability of the existing library. On the other hand, more effort would have to be invested to ascertain and maintain stability and compatibility with GASNet’s large codebase, leaving less resources to experiment with new features.

2. *Building our own Core API implementation from scratch*: This option leaves us with a high degree of flexibility on how exactly to implement the API, and the leeway to depart

---

\(^4\)This limitation can be easily lifted by moving the source identifier into a later word; furthermore, based on the current packet size limit, 12 bits instead of 16 would suffice for the word count. The destination identifier could therefore be extended to up to 20 bits, the equivalent of 1048576 nodes, without major changes.

---

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
</tr>
</tbody>
</table>
Chapter 3. THEGASNet - A heterogeneous GASNet implementation

from the specification when it conflicts with our design goals. However, the larger GASNet user community, including developers of PGAS languages, does not directly benefit from our research, and use of our work by others will likely be very limited when it is made available. It is also highly likely that our new software implementation will not perform as well as the mature reference code.

We decided on the second approach based on the following considerations:

- Larger degree of control: In implementing the pre-defined API from the ground up, we were able to use our limited resources to incrementally build up the features we considered most essential and in exactly the ways that we judged most well-suited to our targeted execution platforms.

- Unavoidable incompatibilities: To accommodate heterogeneity and the inclusion of hardware components, we had to depart at points from an exact implementation of the GASNet specification. In fact, these departures represent some of our most important takeaways for how to make heterogeneous PGAS work. As a consequence, a extension of the library that stays compatible with the existing API would not have been possible, and existing GASNet application code would break anyway if linked with our code. We hope that these conclusions can assist in defining the emerging new GASNet-EX standard [53].

- MicroBlaze code size: By concentrating on the most essential Core API features and realizing them as a bare-metal embedded runtime, we were able to keep the footprint of our MicroBlaze Core API implementation (see Section 4.1) small enough to run credible application code just from on-chip memory, which simplifies the processor system architecture. This would have been unlikely with the GASNet codebase, which is targeted for a host operating system (See Ramp BLUE [41] again).

A well-understood consequence of this approach is that it is not a realistic possibility to adapt established and productively used PGAS languages to our library in the near future. Instead, we plan our own, heterogeneity-aware high-level library (see Section 8.3) which shares many of their properties. Our effort is intended as a technology exploration and not expected to be adopted directly for use by the larger high performance computing community.
3.5.2 Linux-based multi-threaded implementation

The THeGASNet implementations for both the x86-64 and ARMv7 architectures are designed to run as applications under a modern Linux operating system kernel, using the POSIX Pthreads standard to implement multiple threads and the TCP/IP socket interface for Ethernet communication. Therefore, only minimal differences apply between the two platforms:

- Because of the different native word lengths, different macros are used to convert between native pointers for local use and the \texttt{voidp64} type that is used by our library for all remote memory pointers.

- For the communication from CPU to FPGA fabric the x86-64 version uses PCIe, while the ARM version uses the AXI bus. This is not strictly related to the processor architecture, but to our specific target platforms, and other combinations might be used in the future. As both kinds of buses are encapsulated through file system-mapped I/O, code differences are minimal.

Since multicore processors as well as symmetric multi-threading are widely established by now, multiple threads or processes running in parallel are necessary to exploit a modern processor’s performance potential. Despite this added complexity, these features also offer to accommodate task parallelism in implementing a GASNet library. A simple approach to this is implementing a single GASNet node per host and leaving it to the application programmer to exploit thread-level parallelism on that single system. There is ample precedent of this in combining MPI and OpenMP to optimize for both local and distributed parallelization [54]. However, this does add significant complexity to the application programmer’s task in composing both approaches and maintaining correct parallelism. It is therefore desirable to offer the option of running several GASNet nodes on a multi-threaded multi-core processor.

There are effectively two practical implementation approaches for this, running several GASNet nodes in multiple threads inside one process, or running an individual process per GASNet node. J. Duell has closely examined both approaches for implementing Unified Parallel C [55].

The multi-threaded solution naturally shares a virtual address space, providing for easy data copying between nodes. One downside of this shared address space is that thread-based nodes
don’t enjoy memory protection of non-shared areas from accidental or deliberate manipulation by other nodes.

The multi-process solution has to implement similar functionality through POSIX shared memory. The result of Duell’s examination is that performance differences between the two implementations are minimal and inconclusive, however a multi-process solution enables better composability of applications using both GASNet and other distributed libraries like MPI.

Despite giving consideration to this conclusion, we have decided that for our THeGASNet implementation a multi-threaded approach is preferable. In this way, we can benefit from the convenient features provided to us by Pthreads, namely the simpler inter-thread communication and synchronization and the locking of shared resources like the PCIe or AXI interfaces. We can also more easily manage the complete local set of nodes on a machine, simplifying application launch and optimizations like CPU thread affinity.

We consider it an important part of the programming model that a programmer can design an application as if running a single GASNet node in SPMD fashion, including the assumption that an application starts with the \texttt{main()} function. Even if the application might be run as multiple nodes in multiple threads, the code should conform to the single-node model that is the base of the GASNet API specification. This approach also facilitates easier code portability to a strictly single-threaded platform like the MicroBlaze processor.

To avoid writing our own C runtime library to accommodate these properties, we are using a link-time function wrapper mechanism provided by GCC. Figure 3.10 illustrates our multi-threaded approach. When linking the application with GCC, the GASNet library provides a function called \texttt{\_\_wrap\_main()} that will be called by the C runtime library instead of \texttt{main()}. The wrapper function expects two command-line arguments, the local IP address and the name of a configuration file. The configuration file, which is identical for all hosts in the distributed system, holds a single line entry for each host that specifies its IP address, the GASNet IDs of software and hardware (FPGA) nodes on this host, and routing policies on this host. Based on this information, the main wrapper can generate a localized routing table with information for how to reach every node in the complete distributed system. It further makes copies of the command-line arguments and adds extra parameters that specify GASNet properties like node IDs and overall number of nodes in the system.
The wrapper function then creates a number of threads corresponding to the number of local software nodes. Each thread calls an instance of `main()`, each receiving a set of command-line arguments with an individual GASNet node ID.

When each `main()` thread calls `gasnet_init()`, node-specific data is initialized and a separate thread for handler functions is created. This handler thread shares all node-specific data with the application thread. By spawning an independent handler thread, concurrent execution of application and Active Message reception is facilitated.

When `gasnet_attach()` is called the first time, further separate threads for the reception of FPGA data (PCIe or AXI) and TCP/IP messages are created. These threads can be suspended on blocking `read()` calls and will only be activated if an Active Message arrives through the corresponding channel. In this case, the received data is provided to the handler thread corresponding to the destination ID. Active Messages can also be directly routed from TCP/IP to FPGA or vice versa if such routing has been specified in the system configuration file.
If data is communicated between two local nodes, the application thread of the sender will do a direct memory copy to the shared segment of the receiving node and then notify the destination node’s handler thread of the Active Message (for testing purposes, two local nodes can also communicate through the TCP/IP localhost).

The current threading solution requires special treatment of global variables, which have an important function in each node as the synchronization mechanism between application and handler functions. By default, global variables are shared by all threads. When we are spawning multiple threads to implement multiple GASNet nodes, it would break the programming model of independent nodes to have these global variables shared between all of them. GCC provides the \_\_thread keyword, which implements a separate copy of a global variable for each thread if the variable is declared with the keyword. However, this in itself is not useful either, as application and handler functions for the same node are in separate threads. To resolve the issue, our GASNet implementation provides proprietary macros that a programmer can use to group an application’s global variables into a struct type. One struct of this type is then dynamically allocated by each node, and both the application and handler threads can access the struct members through a pointer. This solution still breaks strict compatibility with GASNet since conventional file-scope globals are still not shared between application and handler thread. However, we judge this to be a satisfactory compromise.

3.5.3 Differences to the GASNet Core API specification

Aside from the different handling of global variables required by our multi-threading approach (see Section 3.5.2), our implementation includes a few deliberately introduced distinctions from the GASNet API specification:

1. Any specification of a remote memory address or remote memory segment size uses the type voidp64, which is a typedef referring to a 64-bit unsigned integer. In a homogeneous GASNet system composed of nodes that all share the same architecture, the type void* is correctly translated to the native pointer size. However, all our THeGASNet nodes need to use the maximum pointer size in the distributed system for all remote memory parameters. This applies to Active Message function calls specifying a remote address as
well as to the contents of each node’s segment table, which stores the position and size of the shared memory segments of all nodes.

2. A remote address in our implementation always represents an offset from the start of the shared memory segment of the remote process; in contrast, in GASNet it represents the exact virtual address in the remote process. Every application using original GASNet needs to store a segment table that holds an entry specifying the virtual start address and the size of every other node’s shared memory segment. To implement a distributed data structure like the shared arrays in UPC, every relative access in that structure also needs to be offset with the virtual start address for the remote node. RDMA hardware at the remote node (e.g. an Infiniband NIC) can then read or write the data based on the remote processes’ page table. We considered the implementation and constant access of the GASNet segment table as wasteful for use in FPGAs, where on-chip memory is limited and off-chip memory generally is managed statically and without address virtualization. Therefore, we communicate remote offsets that can be added to the actual start of the shared segment by either THeGASNet hardware components or the THeGASNet software library. Our software library still holds a segment table for code compatibility reasons, but all its start addresses are set to 0. One implication of this change is that local shared addresses that are communicated as handler arguments (e.g. in a request initiating a remote read) need to be changed to offsets first. Another one is that the shared segment size for every software node is assumed to be the same, and the one of FPGA nodes either assumed to be the same size or statically predefined to be a specific size by the software application.

3. In setting up an application with the regular GASNet library, after the initial gasnet_init() call, shared memory is allocated through calling gasnet_attach(). The collective segment address exchange implemented in gasnet_getSegmentInfo() also works as an implicit barrier, only allowing remote memory access after it has been called by every node. Since our implementation does not actually exchange segment data between nodes anymore, it would be necessary to implement a regular global barrier in gasnet_getSegmentInfo() just to avoid remote accesses to unallocated memory.
Furthermore, because _gasnet.getSegmentInfo()_ is also the only way for a GASNet node to learn its own shared segment’s address, it is not possible to initialize the segment with data before other nodes access it, unless the programmer introduces an initial, non-standard barrier to the application.

Thirdly, FPGA hardware-only components with their static, reset-based bootstrap have no equivalent to at least the first two of these functions; however, they might also require initialization of memory before allowing remote memory accesses.

To tackle both the memory initialization issue and the different application launch, we have decided to introduce an explicit barrier, called _thegasnet.SharedMemoryReady()_ that is called by both software and hardware components after they have prepared their shared memory for remote use. While in software this will happen after calling the three functions and preparing the shared memory, in hardware it will be the very first step after memory initialization.

4. We have decided to extend the THeGASNet Core API implementation beyond the scope of the GASNet standard by adding _strided_ and _vectored_ (scatter/gather) memory transfers. The integration of such transfer modes into the Extended API has been suggested by Bonachea in [56] because many computation applications need to communicate in these patterns. We have concluded that such functionality is necessary and useful at the Core API level in the case of our implementation. We suspect that the original rationale for defining these types only at the (unlimited transfer-size) Extended API level was that on processor-based architectures, the packing and unpacking of such data would be done by software anyway, and regular Core API _Long_ messages would suffice for the transfer of assembled packets. However, the purpose of our FPGA THeGASNet components is to manage all complex API tasks outside the computation node (see Section 4.2.2).

The _StridedLong_ message type allows the packaging of multiple data strings of identical size and set apart by an identical stride into a single message payload, and unpack these in a similar way at the destination. In practice, this enables remote copying of a subarea of a 2-dimensional data structure to another 2-dimensional structure. This can also be used to transpose matrix data. We have decided against implementing higher-dimensional
strides in the Core API function as this would cause significant overhead in our hardware implementation and will rarely be justified due to the size limit of individual messages. The Extended API or other higher-level functionality can divide a higher-dimensional strided transfer into 2-dimensional portions with limited effort.

The * VectoredLong message type enables transmissions of several blocks of data of various size and distribution in a single packet. This means the address and size of every block (somewhat inaccurately labeled a “Vector”) has to be specified explicitly. We currently limit the number of source and destination vectors each to 16.

An overview of the differences between GASNet and THeGASNet can be seen in Table 3.2.

<table>
<thead>
<tr>
<th></th>
<th>GASNet</th>
<th>THeGASNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote address type</td>
<td>native void*</td>
<td>voidp64</td>
</tr>
<tr>
<td>Remote addresses</td>
<td>virtual address</td>
<td>offset to shared</td>
</tr>
<tr>
<td></td>
<td>in shared segment</td>
<td>segment start</td>
</tr>
<tr>
<td>Global variables</td>
<td>regular</td>
<td><em>NODE GLOBAL</em> macros,</td>
</tr>
<tr>
<td></td>
<td>ANSI C</td>
<td>struct member references</td>
</tr>
<tr>
<td>Strided and Vector messages</td>
<td>suggested for</td>
<td>part of Core API</td>
</tr>
<tr>
<td></td>
<td>Extended API</td>
<td></td>
</tr>
<tr>
<td>Barrier for shared memory initialization</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

### 3.5.4 Platform-independent Non-GASNet functionality

The THeGASNet API is by definition fully portable between the supported processor platforms. However, as the bare-metal and Linux-based implementations deliver some basic functionality through distinctly different mechanisms, we have also implemented platform-independent macros for these:

- *PRINT()* is a macro that resolves to standard *printf()* on the Linux platforms, but to a significantly more memory-efficient function *printfast()* on the MicroBlaze\(^5\). *WAITKEY-PRESS()* similarly abstracts console feedback to the application.

\(^5\) *printfast()* also circumvents the C runtime’s *stdout* mechanism to enable use of a resource-efficient FSL-connected UART peripheral we designed.
Several `Timer_*()` macros provide performance timing functionality for all platforms at microsecond granularity.

`READ_SHARED()`, `WRITE_SHARED()` and `ADD_TO_SHARED()` enable atomic access and memory fences to read and modify variables shared between different threads (most importantly, the global variables shared between a node’s application and handler threads). GCC provides platform-independent macros for these operations, but the MicroBlaze GCC implementation does not support them (On the MicroBlaze, the simple read and write macros up to the 32-bit native word size resolve to simple accesses; all others execute atomically by masking interrupts for the complete operation).

### 3.5.5 THeGASNet code example

The following code example implements the string copy application from Section 3.3.3 with THeGASNet. Identical code is grayed out, changes are in black and discussed below.

```c
#include <stdio.h>
#include <string.h>
#include "platforms.h"
#include "thegasnet_core.h"

// function prototype for barrier
void barrier();

// GASNet handler function prototypes
#define hID_S0_barrier_increment 140
void S0_barrier_increment(gasnet_token_t token);

#define hID_L0_receivestring 131
void L0_receivestring(gasnet_token_t token, void *buf, size_t nbytes);

// handler function table
static gasnet_handlerentry_t handlers[] =
{
    {
        hID_S0_barrier_increment,  
        (void(*)(void))S0_barrier_increment
    },
    {
        hID_L0_receivestring,
```
Chapter 3. TheGASNet - A heterogeneous GASNet implementation

```c
(int main(int argc, char **argv) {
    NODE_GLOBALS_INIT(shared, .barrier_cnt=0, .received=0);
    gasnet_seginfo_t *segment_table_app;
    void* mysharedmem;
    size_t sharedmem_size = 0x100000; // 1MB byte shared mem to allocate
    unsigned int nextnode;
    /* nextnode_sharedmem is not required */
    ptrdiff_t string_offset = 1024; // position in shared memory
    char *teststring = "Mr.Watson, come here, I want to see you.";

    // gasnet start-up
    gasnet_init(&argc, &argv);
    gasnet_attach(handlers, sizeof(handlers)/sizeof(gasnet_handlerentry_t),
    (uintptr_t)sharedmem_size, (uintptr_t)0);

    segment_table_app = malloc(gasnet_nodes() * sizeof(gasnet_seginfo_t));
    gasnet_getSegmentInfo(segment_table_app, gasnet_nodes());

    // find nextnode, shared memory segments
    mysharedmem = (void*)segment_table_app[gasnet_mynode()].addr;

    nextnode = (gasnet_mynode() + 1) % gasnet_nodes();

    // GASNet is initialized
    PRINT("Node %d ready.\r\n", gasnet_mynode());
    thegasnet_sharedMemoryReady(0);

    // First node: Send off string
    if (gasnet_mynode() == 0)
        gasnet_AMRequestLong0(nextnode, hID_L0_receivestring,
```
teststring, strlen(teststring)+1, (voidp64)string_offset);

// All nodes: Waiting for message receive; could do other stuff meanwhile
while(!READ_SHARED(shared->received));

char *rcvstring_local = READ_SHARED(shared->rcvstring);
// All other nodes: Send string on
if (gasnet_mynode() != 0)
gasnet_AMRequestLong0(nextnode, hID_L0_receivestring, (void*)rcvstring_local, strlen(rcvstring_local)+1, (voidp64)string_offset);

PRINT(“Node %d received at address %p: %s\n”, gasnet_mynode(), rcvstring_local, rcvstring_local);
barrier();
PRINT(“Node %d done.\n”, gasnet_mynode());
barrier();
gasnet_exit(0);
return 0;
} // end of main()

void barrier() {
USE_NODEGLOBALS(shared);

if (gasnet_mynode() == 0) {
// wait for messages from all others
while(shared->barrier_cnt < gasnet_nodes()-1);
shared->barrier_cnt = 0; // reset

// send complete messages to all others
for(int t=1; t < gasnet_nodes(); t++)
    gasnet_AMRequestShort0(t, hID_S0_barrier_increment);
}
else {
// Send message to 0
gasnet_AMRequestShort0(0, hID_S0_barrier_increment);

// wait for complete message from 0
while(shared->barrier_cnt == 0);
shared->barrier_cnt = 0; // reset
}
// GASNet handler functions
void S0_barrier_increment(gasnet_token_t token) {
    USE_NODEGLOBALS(shared);
    ADD_SHARED(shared->barrier_cnt++, 1);
}

void L0_receivestring(gasnet_token_t token, void *buf, size_t nbytes) {
    USE_NODEGLOBALS(shared);
    shared->rcvstring = (char*)buf;
    shared->received = 1;
}

The differences in the multi-platform THeGASNet implementation lead to the following modifications in the example application:

- In line 65, a mandatory synchronization call to `thegasnet_sharedMemoryReady` has replaced the optional barrier used before. This call is being served by both software threads and FPGA components introduced later and signals that all preparations for beginning of the application have been completed in software and hardware components.

- Because remote addresses are only expressed in offsets to the start of the shared segment, we don’t need the pointer `nextnode_sharedmem` anymore (Line 45). When initiating a `Long` message as in lines 71 and 81, we are only specifying the offset. However, the offsets need to be typecast into the `voidp64` type to be platform-independent.

- As described in Section 3.5.2, we are using a custom mechanism to substitute regular global variables with a structure that can be referenced by the application and handler threads of the same node. In line 31 and following, the macros `START_NODE_GLOBALS` and `END_NODE_GLOBALS` bracket our global variable declarations. Since the macros implement a C `struct` type definition, we cannot initialize the variables in this place. Instead, the `main()` function starts with another macro, `NODEGLOBALS_INIT` in line 39, that allows initialization of the struct members and also initializes and names a local pointer to the struct; in our example the pointer is named `shared`. Any function that
accesses global variables also needs to initialize a struct pointer with the
\texttt{USE\_NODE\_GLOBALS} macro, as seen in lines 96, 120 and 127.

- As a consequence, all accesses to global variables turn into pointer references to struct
  members of the form \texttt{structptr\rightarrow globalvar}.

- We further need to assure memory consistency for data accessed from different threads
  on all supported platforms. We are therefore reading shared globals that are modified
  by other threads with the \texttt{READ\_SHARED} macro in lines 74 and 76, which imposes a
  memory fence. Furthermore, the read-modify-write action in line 122 is made atomic
  with the \texttt{ADD\_SHARED} macro (While we have not used explicit atomicity support in
  the standard GASNet example to keep the example simple, similar mechanisms can be
  required, depending on the specific implementation/conduit).

- Lastly, the \texttt{printf} instructions have been replaced by \texttt{PRINT} macros that resolve to a
  print function with low memory footprint on the MicroBlaze.
Chapter 4

THeGASNet implementation in the FPGA fabric

An essential part of this work are components that accommodate usage of the GASNet API when moving portions of a software-only application into the FPGA hardware. The basic “movable” unit in our software-hardware migration philosophy is the GASNet node. When isolating a software section or component to move it to hardware, this complete component should already be interacting with the rest of the system as a unique GASNet node. The practical design step of migration then involves not instantiating this host software node anymore, but instead instantiating an on-FPGA node that communicates with all other GASNet nodes in the same way that the original software node did. Figure 4.1 illustrates this: On the left side, four software nodes are executing on a host CPU. On the right side, software node 3 has been substituted by a hardware node with the same ID number on the FPGA. All nodes are still operating as part of the same logical THeGASNet node network, as demarcated by the dashed line. Besides making the actual hardware component available, this migration involves changing routing policies in the system to relay all Active Messages addressed with the specific GASNet node ID to the new hardware node, and vice versa.

Enabling a GASNet node in the FPGA fabric to keep up the previous host software node’s communication patterns appears reasonably trivial when it is implemented on a MicroBlaze processor. A MicroBlaze version of the THeGASNet library described in the previous chapter
could use any available on-chip networking infrastructure to send or receive data the same way that the x86-64 and ARMv7 versions do it using PCIe or IP, with differences limited to the exact device to read from or write to.

However, should the GASNet node on the FPGA be a custom computation core replacing a specific part of the application, the designer of this core would inherit a much more complex task. Since the custom core cannot run software, but only execute its specific computation task, the core designer needs to build the complete functionality to send and receive Active Message data packets as previously specified, including the reading and writing of appropriate data and the packing and unpacking of header data. Obviously, this should not be the responsibility of the application core designer, which is why THeGASNet provides a hardware core called GAScore for this task.

The THeGASnet software library provides pre-implemented communication functionality to software application programmers. In a similar fashion, GAScore provides equivalent communication functionality to a hardware application designer. Moreover, to make the migration step from software to hardware even more intuitive, the THeGASNet software implementation for the MicroBlaze also utilizes GAScore for its GASNet communication. As illustrated, it amounts to no more than a direct substitution to replace the MicroBlaze running a software GASNet node in Figure 4.2a with a custom core in Figure 4.2b (Section 4.2.2 describes the connectivity in more detail).
4.1 MicroBlaze THeGASNet software implementation

THeGASNet for the MicroBlaze soft processor operates from a very different starting point vs. the implementations for the x86-64 and ARMv7 processor platforms. A major motivation for a different implementation approach is the high resource cost of running Linux on the MicroBlaze, namely implementing a complete Memory Management Unit and requiring extensive use of off-chip memory for bootup and storage. MicroBlaze also does not have any built-in support for multithreading, although software context switches can of course be accomplished with the help of interrupts.

We have therefore decided to implement GASNet programs on MicroBlaze as a bare-metal application running only as a single GASNet node. As a consequence, it can be realistically used with on-chip instruction memory only, which currently is limited to 128KBytes per processor. Our statically linked THeGASNet library is smaller than 40KBytes, leaving sufficient space for non-trivial applications. The Harvard-architecture MicroBlaze therefore only requires an off-chip memory bus connection for application data.
Since a significant function of the MicroBlaze is to provide an easier migration path to custom hardware processing cores, operating it as a single node is appropriate. This constraint also implicitly limits the required software functionality, as most essential library functionality is provided by the hardware GAScore described in Section 4.2.2.

Since most memory transfer capability is implemented by hardware, THeGASNet library code on the MicroBlaze focuses on transmitting Active Message parameters to the GAScore through a Fast Simplex Link (FSL), a Xilinx-specific 32-bit FIFO channel directly supported by the MicroBlaze instruction set. The parameters received by GAScore correspond closely to the original GASNet function arguments, so that the software becomes a thin translation layer from function call to hardware interface.

After completing the direct memory access portion of an incoming Active Message, GAScore can send a handler function request with arguments back to the MicroBlaze by FSL. Incoming handler requests are handled by the MicroBlaze library through interrupts, thereby enabling logically concurrent execution of application and Active Message handling. To ensure correct operation between handler interrupts and application, the following conditions need to be respected:

1. Interrupts must be disabled when initiating a Request message from the application. Since a handler function can initiate Reply messages, no other Active Message can be in the middle of parameter transmission when a handler function is being executed. THeGASNet takes care of appropriate interrupt deactivation.

2. Global variables shared between application and handler functions need to be declared volatile to preclude compiler optimizations from removing certain operations on them. When using the macro mechanism described for global data structures in multi-threaded implementations, all included globals will be declared volatile implicitly.

Since the GASNet application is started at boot, not dynamically by an operating system, no command-line arguments can be given to the application. To enable the use of application-specific parameters across all platforms, the MicroBlaze implementation of THeGASNet also uses a link-time wrapper function for main(). In this case, the function’s purpose is to generate
argc and argv from preprocessor \#defines ARG1 up to ARG5 specified to GCC at compile-time. The application’s main() function is then called with these generated parameters.

4.2 THeGASNet hardware components

The following sections describe the various hardware components we provide to support implementing GASNet applications in the FPGA fabric. They include components required to use the GASNet API (GAScore), optional components to help design productivity of custom hardware cores (PAMS) and on-chip networking components (NetIf) that could be substituted because THeGASNet abstracts network specifics from the computation hardware.

4.2.1 NetIf communication

On-chip communication between FPGA GASNet nodes is provided through the NetIf interface introduced in previous work [18]. Its basic communication channel is the Fast Simplex Link, a FIFO that can be implemented with Xilinx FPGA primitives at very low resource cost. MicroBlaze processors have native interfaces to send and receive data through 32-bit wide FSLs with an extra control bit. Components in different clock domains can be connected with an asynchronous FSL variant at the cost of increased latency.

The NetIf uses 32-bit wide FSLs with an added control bit that indicates a packet header when asserted. A packet header is a single 32-bit word containing an 8-bit source address, an 8-bit destination address and a 16-bit packet size field indicating the number of 32-bit words to follow (we described this header as part of a general Active Message packet in Section 3.4.3). Packets do not contain checksums or any other kind of redundancy under the assumption that on-chip communication is error-free. As stated in its name, an FSL is a uni-directional channel. The NetIf uses two FSLs between any two nodes to create independent duplex communication.

A typical on-chip system using the NetIf can be seen in Figure 4.3. Each device in the system that needs to send or receive data packets employs its own NetIf router. This includes CPUs or other processing nodes as well as interfaces for off-chip packet communication. The NetIf makes use of an FPGA’s abundant routing resources by implementing a fully connected topology (each double-sided arrow symbolizes a bi-directional pair of FSLs). As a consequence,
NetIf routers are very simple in structure and can route packets with a single cycle of latency. A router init unit supplies all routers with a routing table at system startup. When the connected host device sends a packet to the router, the routing table specifies which output leads to a specific destination address. Incoming packets are passed on to the local device. By default, packets arriving in the same cycle are prioritized by channel number.

The choice of NetIf as our on-chip network is motivated by its small resource requirements and low latency. Since the universality of the Active Message packets introduce an overhead in both size and generation latency compared to application-specific solutions, we would like to not add unduly to this latency through use of a complicated routing scheme. Nevertheless, more investigation is necessary to prove that this simplicity does not throttle bandwidth.

Table 4.1 lists the resource utilization for the NetIf infrastructure for networks with five, seven and eleven NetIf routers. These numbers are motivated by the number of hardware GASNet nodes supported by our reference platform introduced in Chapter 5. MapleHoney is a multi-FPGA platform with two memory channels per FPGA, each of which supports one,
Chapter 4. THEGASNet implementation in the FPGA fabric

two or four connected GASNet nodes. Furthermore, three NetIfs are required for off-chip communication in the form of one PCIe connection and two links to neighbouring FPGAs.

Table 4.1: Post-map resource utilization for various NetIf configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 NetIfs</td>
<td>1099 (0.16%)</td>
<td>2309 (0.67%)</td>
<td>1 (0.16%)</td>
</tr>
<tr>
<td>7 NetIfs</td>
<td>2076 (0.30%)</td>
<td>4722 (1.37%)</td>
<td>1 (0.16%)</td>
</tr>
<tr>
<td>11 NetIfs</td>
<td>4986 (0.73%)</td>
<td>12313 (3.58%)</td>
<td>1 (0.16%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

As the table shows, NetIf resource use on the large Virtex-6 FPGAs is relatively small, however the fully connected topology scales badly to larger number of nodes, as should be expected. We don’t currently envision much larger numbers of GASNet nodes per FPGA since they imply larger numbers of independent memory channels. However, the exploration of other NetIf topologies is possible; for further examination, we refer to previous work on the subject [57][58].

Our GAScore architecture is not tied to a specific on-chip network, and if larger buffer capacities, virtual channels or different network topologies lead to better application performance, changing the network architecture is possible without changing the GASNet semantics. The only requirement posed by GAScore is that the network must be able to receive and deliver contiguous packets of a size of at least eight KBytes plus header information. The availability of hardened on-chip networks on FPGAs as examined in [59] appears to be a natural fit for the architectures THEGASNet enables.

4.2.2 GAScore: Remote DMA support

The central component that enables GASNet operation in the FPGA fabric is called GAScore (Global Address Space core). It’s essential function is as a Remote Direct Memory Access unit, combining a memory bus master with a network interface to copy data across the network without host device involvement. It can be controlled through Fast Simplex Links by a MicroBlaze processor or a custom hardware core as previously shown in Figure 4.2. Each TheGASNet hardware node will be composed of exactly one computation core and one GAScore, who connect to their own private memory (PGAS assumes physically distributed memories, and implements
the *shared* programming model through the Remote DMA functionality).

Figure 4.4 shows the high-level internal structure of a GAScore and its interaction with other components. A GAScore is connected to a NetIf router by bi-directional FSL channels and to a memory or memory controller through an AXI bus. Furthermore, it is connected to a processor or computing core by two pairs of FSL channels for Active Message and handler function requests respectively. GAScore operates two largely independent sections, one for the transmission and one for the reception of Active Messages. The only (non-blocking) connection is a buffer that holds the token information required by the *Request-Reply* semantics.

![External memory controller](image-url)

**Figure 4.4: Overview of GAScore internals**
Figure 4.5: GAScore memory read and Active Message transmission unit

**Active Message transmitter**

Figure 4.5 illustrates the structure of GAScore’s Active Message transmission mechanism. A host device like a MicroBlaze processor requests an Active Message by sending the parameters for an Active Message function call through a Fast Simplex Link to the GAScore. The format for these requests is similar to the platform-independent message formats discussed in Section 3.4.3. As an example, Table 4.2 shows the request parameters for sending a *Long* type message. The major differences are:

- Some initial parameters are sorted in a different order to optimize for processing the request with minimum latency.

- The request data includes parameters specifying the source memory, which are irrelevant for the destination and therefore not included in the exchanged packet.

- The payload is not included yet, with one exception: If the *FIFO* bit is set in the *Type* field, data is not read from memory, but attached to the header and parameters relayed through the FSL.
Table 4.2: Active Message request parameters - Long message

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Destination/Token (16)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Source (16)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
</tr>
<tr>
<td>Source address, lower half (32)</td>
</tr>
<tr>
<td>Source address, upper half (32)</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
</tr>
<tr>
<td>0..512 words memory payload (32)</td>
</tr>
</tbody>
</table>

Based on the message parameters, an appropriate Active Message packet is synthesized. In the case of Short messages or messages with the FIFO type bit set, no data is requested from memory. All other types require data from memory. Based on the parameters specifying the location, size and distribution of the source memory, read requests go through several processing stages:

1. Requests for contiguous blocks of memory (chunks) are generated. In the case of a standard Long message, this is only one chunk. For the LongVectored type, each particular vector represents one chunk. In the case of LongStrided messages, one chunk request is generated for each contiguous string of memory.

2. Based on the specified maximum burst size for the AXI bus, chunks are split up into burst-size requests transmitted to the AXI bus’ Read Address Channel. The burst information is also relayed to the components processing the returned data bursts.

3. Based on the burst information, incoming burst data from the AXI Read Data Channel is reconfigured so that the first requested byte in the burst is aligned to the word size used in the Active Message packet.

4. Data from individual bursts is concatenated into a contiguous data payload. For this purpose, burst data is realigned to fill the payload without any byte gaps. Packet capacity is thus maximized and re-distribution at the destination in a different pattern is facilitated.
5. In the transmission unit, the data payload is attached to the packet head holding message parameters and handler arguments. Finally, the packet is sent out to the local NetIf.

As soon as a message has been released to the network, a data word is sent back to the GAScore host through the Active Message return FSL. All Active Message calls except the ones designated Async wait for this signal before returning and continuing operation. The return indicates that the source memory required by the message has been read and can be modified again by the application.

**Active Message receiver**

Figure 4.6 details how an incoming Active Message is processed by the GAScore. The sequence of actions depends on which one of the major types the message represents:

- **Short** messages holding no data payload are directly relayed to the GAScore’s host device in the form of a handler function call. GAScore transmits all parameters required by the
appropriate handler function prototype through an FSL. An example of the data sent for a Short type handler call can be seen in Table 4.3.

- Medium messages are also relayed directly to the host device. A unique property of using GAScore is that the content of Medium messages is not copied into a temporary memory buffer at the destination, but instead attached to the handler call parameters flowing through the FSL. This accommodates the difficulty of dynamic memory management for FPGA hardware and small embedded systems. As a side benefit, it establishes a direct data channel between nodes (similar to TMD-MPI [18]) as another communication option.

- Long, Strided Long and Vectored Long messages process the required memory writes before releasing the handler function request from a buffer. This meets GASNet semantics in which new data always has to be present in memory by the time of the handler function execution.

Writing data payload to memory also requires a multi-stage processing sequence:

1. According to the parameters for the payload distribution, destination addresses and sizes for contiguous chunks are generated.

2. Contiguous chunks need to be split into burst requests to the AXI Write Address Channel. The number of burst requests are counted to keep track of the number of outstanding data bursts.

3. The data payload is split into separate chunks according to the chunk information generated before.

4. Data chunks are word-aligned to the AXI buses word width.
5. Aligned chunks are re-aligned inside the bus word to start at the byte offset corresponding to the destination address and written to the AXI Write Data Channel. Data does not have to be split up into bursts per se, however the indicator bit for the last data word in each burst needs to be asserted appropriately.

6. The AXI bus indicates a completed burst through the AXI Write Response Channel. Based on this information, the burst count is decreased until all outstanding bursts have arrived. At that moment, the handler function call is released to the FSL.

After the handler function has been completed on the GAScore’s host device, the host sends back the token that was part of the handler call, thereby freeing it for use in a future handler call.

**Token buffer**

The only component in Figure 4.4 that connects the transmission and reception parts of the GAScore is the Token Buffer. This component is responsible for implementing GASNet’s Request-Reply semantics: An initial Active Message request from a node A to a node B must, at the most, result in a single Active Message reply from B to A. This mechanism is formalized by withholding the identifier of the source node from the handler function; it is instead stored by GASNet itself and linked to a Token which is provided to the handler function. If the handler function requires the sending of a reply, it specifies the token instead of a specific node and GASNet retrieves the appropriate node address.

The hardware implementation of this mechanism requires two storage components, a FIFO and a dual-ported RAM. The FIFO is loaded at system startup with 512 unique token values. When a handler call is synthesized by the Active Message reception unit, a token is read from the FIFO and embedded in the handler parameters. Concurrently, the source node identifier of the incoming message is stored in the dual-ported RAM at the address specified by the token value.

If the handler function initiates a reply Active Message, the token in the request is used to

\(^1\)Since the handler call triggered by a Reply message is identical to the one for a Request, such a handler call receives a token, too. Consequently, a continuing ping-pong of Reply messages is technically possible, but the GASNet standard explicitly forbids this.
reference the node identifier in the RAM, which can then be read and used as a destination identifier.

When the handler function has completed execution, it signals completion by returning the token to the GAScore through the Handler return FSL. GAScore feeds the token back into the FIFO for re-use.

The use of the dual-ported RAM and the separate producer/consumer roles for the FIFO enable fully concurrent operation of the reception and transmission units. However, the system can be stalled if 512 tokens are taken from the FIFO without being returned, representing 512 handler functions being called without returning correctly. We consider it a reasonable assumption that this circumstance would constitute an error or bad practice by the application designer.

Node configurations using GAScore

A key GAScore feature is that it can be used both by software and hardware implementations of a GASNet application. Figure 4.7 shows a GASNet node running in software on a MicroBlaze CPU. An application is running on the MicroBlaze processor, using the GASNet single-threaded software library as a communications runtime. The processor is connected to the GAScore by the two FSL pairs for Active Message requests and handler function requests. As visible on
the left side, the MicroBlaze will also likely be connected to other components like local code and data memory and peripherals like UARTs and timers, some of which might be additional address-mapped slaves on the processor’s AXI bus.

MicroBlaze supports FSL access through native assembler instructions. Because the parameter format and GAScore functionality have been so closely modeled on the GASNet specification, translation from Core API function calls to FSL writes is very simple and efficient. The same is true for incoming data on the Handler call FSL, which triggers an interrupt that can call the handler function with the incoming data.

GAScore and MicroBlaze are both masters sharing an AXI bus that gives access to external DRAM through a memory controller. Any details about the external memory interface are abstracted from the processor and core through the AXI bus. Consequently, it is possible to share any type of memory that can be accessed through AXI, including on-chip BlockRAM. Neither bus master privately caches any memory, so no coherency mechanism is required. This is likely detrimental to the processor’s performance, however MicroBlaze performance of GASNet applications is not our primary concern because the processor’s main purpose is as an intermediate platform for software-to-hardware migration.

In Figure 4.8, the MicroBlaze processor has been replaced with a custom processing hardware that supposedly will fulfill the same function as the software application. To function as a
drop-in replacement, it needs to replicate the processor’s bus connectivity to external memory and to the GAScore (in most cases, it will likely not require the processor’s other peripherals). It must be able to act as an AXI master, generating (burst) reads and writes as required by the computation. For GASNet communication, it needs to be able to generate requests for Active Message transmission as an FSL master and receive confirmation of their completion as an FSL slave\(^2\). Furthermore, it needs to be able to receive handler function requests, initiate appropriate handler functionality and return the received token to the GAScore for re-use. As these interactions with GAScore can impose considerable complexity, we are introducing a hardware solution to handle them in the following section.

Table 4.4 shows the resource utilization for a single GAScore on our reference platform. In each category, GAScore uses less than a percent of resources on a large Virtex-6 FPGA, making it a very affordable addition to most designs.

<table>
<thead>
<tr>
<th></th>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAScore</td>
<td>1456 (0.21%)</td>
<td>2610 (0.76%)</td>
<td>5 (0.79%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

\(^2\)Completion means that the specified memory has been read and the message has been sent off; GASNet’s asynchronous semantics do not include a mechanism for confirmation of reception except an explicit Reply type message. The network beyond the GASNet node is assumed to be reliable in transmitting the message.
4.2.3 Programmable Active Message Sequencer

As described in the last section, substituting the MicroBlaze processor with a custom hardware solution for the application does not just require replacing the computing functions, but also the software’s interaction with GAScore. The simplicity of the GASNet Core API and its straightforward translation into hardware parameters do not impose undue complexity on this task, but a non-trivial control logic will be required for most applications.

As the design patterns required here will be similar for a wide range of computation applications, we have implemented a small, flexible domain-specific processor for these common tasks, which we call *Programmable Active Message Sequencer* (PAMS). As pictured in Figure 4.9, a PAMS acts as the direct interface to the GAScore channels on the one side and the computation core on the other side. It replaces the need to design an application-specific state machine for each computation core. At the same time, its functionality is focused on coordinating the sequence of computation core operations, communication and synchronization. Consequently, it fulfills these tasks faster and with less memory requirements than a universal processor.

As a programmable state machine that manages control flow and external communication for a computation core, PAMS shows similarities to the concept of *Control Threads* introduced with CoRAM [45]. They both share the capability to sequence computation and external communication for the application core. However, CoRAM’s control threads are responsible

![Figure 4.9: GASNet node with custom processing hardware and PAMS using GAScore](image-url)
for providing the computation core’s local memory with data from external memory as well as
for transporting data back to main memory. As such, they abstract external memory away for
the application core. Application cores using THeGASNet manage their own external memory
access through AXI. They instead use PAMS to initiate data transfers from their local external
memory to other memories through GAScore. THeGASNet abstracts away the communication
infrastructure and device topology for the application core and therefore provides scalability
across larger networks and clusters.

The PAMS is a small domain-specific controller that executes a limited set of machine code
instructions to control the GAScore communication and synchronize it with core computations.
A structural overview of the PAMS can be seen in Figure 4.10. It has the following features:

- PAMS code can be loaded and re-loaded into the Code RAM through specialized Active
  Messages of type Medium (recall that this type’s data payload is channeled directly into
  the GAScore’s host, which is now the PAMS).

- Message counters (MessageCtrs) can be configured to count messages with a specific
  handler ID and indicate when a threshold is reached. This is, for example, useful for
  barriers.

- Transfer counters (DataCtrs) can be configured to count the data that messages of a
  specific handler ID have written into memory. They also have a configurable threshold.
  This is useful for a distributed barrier behaviour in which synchronization is achieved by
  having received the expected amount of data from other nodes.

- The sequencer can wait for a specific timer threshold. This is especially useful for imple-
  menting benchmarks that use the timers to measure communication delays.

- The timer can be modified by a particular offset. This is useful to synchronize timers in
  multiple connected FPGAs that do not come out of reset in the same cycle.

- When a message has arrived, the current timer is copied into the ArrivalTime register.
  This value is useful for benchmarking purposes.
Limited arithmetic capability exists with four 32-bit accumulators than can be modified by data from various sources.

Based on accumulator operations, conditional branches can be executed.

The sequencer can send control signals to the custom hardware.

Correspondingly, PAMS can wait on control inputs from the custom hardware.

Data can be transmitted from and to the computation core.

All possible wait conditions can be configured into one wait instruction, so that execution continues as soon as all conditions are met.

Finally, the sequencer can write Active Message requests to GAScore that include attached arguments from PAMS code, from the accumulators and from the computation core.

A complete overview of the PAMS instruction set is available in Appendix D.1.

As a simple example for tasks the PAMS can fulfill, Figure 4.11 shows pseudocode for a common task, implementing a simple barrier in a 16-node system (An example with actual...
Chapter 4. The GASNet implementation in the FPGA fabric

PAMS instructions can be found in Section 4.2.4. There are two different code versions, one

```c
/* PAMS code for node 0 */
set_timer_threshold(STARTTIME);
wait_for_timer();
set_msg_ctr0(barrier_call,15);
wait_for_msg_ctr0();
send_AM(barrier_done,1);
send_AM(barrier_done,2);
send_AM(barrier_done,3);
    [...] 
send_AM(barrier_done,13);
send_AM(barrier_done,14);
send_AM(barrier_done,15);

/* PAMS code for nodes 1-15 */
set_timer_threshold(STARTTIME);
wait_for_timer();
send_AM(barrier_call,0);
set_msg_ctr0(barrier_done,1);
wait_for_msg_ctr0();
```

Figure 4.11: Example PAMS code

for node 0 and one for all the other nodes. For benchmarking purposes, all nodes are primed
to start at a predefined time. All nodes except 0 send a `barrier_call` message to node 0, and
then wait for a `barrier_done` message. Node 0 waits for 15 `barrier_call` messages (assuming a
16-node system) and then sends one `barrier_done` to every node. Each node automatically logs
the arrival time of the last message in their `ArrivalTime` register, so that it can later be read
for benchmarking purposes.

The Programmable Active Message Sequencer’s functionality has continuously grown with
the requirements of different applications. The current version is sufficient for the applica-
tions demonstrated in Chapter 6, however an enhanced version has been introduced in recent
work [50]. We will discuss further possible improvements in Section 9.1.

Table 4.5 shows the resource utilization for a single PAMS compared with an area-optimized
`MicroBlaze` processor. As can be seen, our custom design in its current state already requires
more area than a small general-purpose embedded processor. It is not trivial to completely
replace the current PAMS functionality by using of a common processor and it would certainly
incur larger latencies. However, we suggest further investigation of such a replacement, as
an established microprocessor would come with advantages like a full compiler and debugging toolchain. Further development of PAMS [50] also suggests increased similarity of requirements with a general-purpose processor.

Table 4.5: Post-map resource utilization, PAMS vs. MicroBlaze

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAMS</td>
<td>1192 (0.17%)</td>
<td>1472 (0.43%)</td>
<td>1 (0.16%)</td>
</tr>
<tr>
<td>MicroBlaze 8.40.a (area-optimized)</td>
<td>689 (0.10%)</td>
<td>987 (0.29%)</td>
<td>1 (0.16%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

### 4.2.4 Code example: THeGASNet with PAMS code generation

The following code example demonstrates how a THeGASNet software node would generate code for PAMS modules in a heterogeneous node configuration. Building on the previously used string copy example, we assume that the second half of the GASNet nodes in the system are FPGA computation cores that use PAMS for their communication control. The code of the software nodes does not change except for an extra segment on node 0, which generates the code for all PAMS modules and programs them by sending the code to them in Medium Active Messages. The new code is again shown in black and only the surrounding code lines in grey are shown to indicate the new code segment’s position in the complete application.

```c
55
56 gasnet_getSegmentInfo(segment_table_app, gasnet_nodes());
57
58 // find nextnode, local shared memory segment
59 mysharedmem = (void*)segment_table_app[gasnet_mynode()].addr;
60 nextnode = (gasnet_mynode() + 1) % gasnet_nodes();
61
62 #define PAMSCODESIZE 1024
63 #define hID_S0_sharedMemoryReady 127
64 #define hID_M0_pams_codeload 64
65
66 // PAMS code generation and programming
67 if (gasnet_mynode() == 0)
68 {
69     uint32_t pamscode[PAMSCODESIZE];
70     int first_FPGA_node = gasnet_nodes()/2; // upper half is FPGA nodes
```
for(u=first_FPGA_node; u<gasnet_nodes(); u++) // for all FPGA nodes
{

    // clear code memory
    for(w=0;w<PAMSCODESIZE;w++)
        pamscode[w] = 0;

    int Node0 = 0;
    int ThisPAMS = u;
    int Next = (u+1) % gasnet_nodes();
    int pcaddr = 0;

    // INIT
    pamscode[pcaddr++] = PAMS_WAITFOR(0,0,0,0,PAMSOP_CODELOAD);
    pamscode[pcaddr++] = PAMS_SETNODEID(ThisPAMS);

    // INIT READY barrier
    // prepare wait for 1 message of type sharedMemoryReady in message counter 0
    pamscode[pcaddr++] = PAMS_MSG_THR(hID_S0_sharedMemoryReady, PAMSOP_MSGCTR0);
    pamscode[pcaddr++] = 1;
    // send one message of type thegasnet_sharedMemoryReady
    pamscode[pcaddr++] = PAMS_TO_AMSEND(1, PAMSSRC_CODE, 2);
    pamscode[pcaddr++] = PAMSF_SHORT | PAMSF_ASYNC | Node0;
    pamscode[pcaddr++] = ((hID_S0_sharedMemoryReady)<<16) | ThisPAMS;
    // wait for message counter 0
    pamscode[pcaddr++] = PAMS_WAITFOR(0, PAMSOP_MSGCTR0, 0,0,0,0);

    // STRING RECEIVE
    // prepare wait for 1 message of type receive_string in message counter 1
    pamscode[pcaddr++] = PAMS_MSG_THR(hID_L0_receivestring, PAMSOP_MSGCTR1);
    pamscode[pcaddr++] = 1;
    // wait for message counter 1
    pamscode[pcaddr++] = PAMS_WAITFOR(0, PAMSOP_MSGCTR1, 0,0,0,0);

    // STRING SEND - LongAM
    pamscode[pcaddr++] = PAMS_TO_AMSEND(1, PAMSSRC_CODE, 7);
    pamscode[pcaddr++] = PAMSF_LONG | PAMSF_ASYNC | Next;
    pamscode[pcaddr++] = ((hID_L0_receivestring)<<16) | ThisPAMS;
    pamscode[pcaddr++] = strlen(teststring)+1; // nbytes
    pamscode[pcaddr++] = string_offset; // srcaddrL
    pamscode[pcaddr++] = 0; // srcaddrH;
    pamscode[pcaddr++] = string_offset; // dstaddrL
    pamscode[pcaddr++] = 0; // dstaddrH;

    // CORE FUNCTION
The new code segment is placed below the calls to the GASNet startup functions and above the barrier that synchronizes the application start across all nodes. In this case, a single
software node (node 0) generates the PAMS code for all FPGA nodes in a loop. Each loop iteration begins by clearing an array named \texttt{pamscode} that will hold the code before sending it off and by resetting an array address iterator named \texttt{pcaddr} to 0. Code is written with the help of preprocessor macros from \texttt{pamscode.h} which effectively look like slightly more verbose assembler mnemonics and which will generate the appropriate 32-bit opcodes.

The first code memory address is filled with a wait instruction that waits for code to be loaded into the PAMS remotely (Line 84). This is the default state of PAMS code memory after reset; execution continues at the next address; the wait instruction is therefore only required to prepare the next program to be loaded into the PAMS (a finished PAMS program loops back to code address 0, hitting the wait condition). In line 85, the first executed instruction of our code sets an internal register with the ID of this node. This information is given to the computation core and can also be used as an input on various operations.

From this point on, the PAMS is supposed to show the same communication behaviour as a software node does. The core’s first communication action beginning in Line 87 is to participate in the \texttt{thegasnet\_sharedMemoryReady} barrier. Therefore, the code first prepares a message counter to count arriving Active Messages with the handler ID \texttt{hID\_S0\_sharedMemoryReady} and to set its \textit{wait threshold} at 1 message. Only after that, the PAMS writes one request for a short message to the GAScore. The message will use the same handler ID and be sent to node 0. With the message sent off, the PAMS starts waiting for the message counter to hit its threshold, meaning that the synchronization message has arrived from node 0.

The segment starting at line 98 sets up a similar wait mechanism, however this time the core will wait for the \textit{Long} message with handler ID \texttt{hID\_L0\_receivestring} to arrive, signaling that the GAScore has copied the string into local memory. Like the software nodes, beginning in line 105 the PAMS will initiate the remote copy of the string to the next node. As our computation core’s shared memory is assumed to begin at memory address 0, both the source address and destination offset specified are both \texttt{string\_offset}.

In our exceedingly simple GASNet example, the only “computation” being executed on the received data is the printout of the string. Starting in line 115, we are writing a pulse to the application core indicating that computation can begin (i.e., the core can do whatever it does to replace the printout). As no other communication or synchronization will happen concurrently,
the PAMS then just waits for a pulse from the computation core signaling the end of operations.

The code finishes with two more barrier operations that mirror the ones in the software version, this time using the `hID_S0.barrier_increment` from the application-side `barrier()` function. After the second barrier, software nodes will call `gasnet_exit();` the PAMS will loop back to code address 0 and wait to be reloaded with a new program.

With the array `pamscode` now filled with the correct opcodes, it will be sent off to the appropriate node through a Medium Active Message with the handler ID `hc_M0.pams_codeload` in line 148. The reserved handler ID will trigger the PAMS to reload its internal code memory with the message’s payload, then release the above-mentioned wait condition so that execution will begin.

In our simple example, the only thing changing between iterations of the code generation loop is the node’s ID, however, because code is generated individually for each node, it could be tailored in various ways, for example involving different communication patterns for different nodes.

After the PAMS code generation, the software continues unchanged. In line 155, the software nodes call the initial barrier `thegasnet_sharedMemoryReady()`, as will the FPGA components.
Chapter 5

Implementation platforms

Our THeGASNet software and hardware components are targeted to be highly portable to
different software and FPGA platforms. In this section, we describe the two systems that
we have used for the implementation of sample applications, a self-built heterogeneous cluster
based on Xilinx Virtex-6 FPGAs and x86-64-architecture PCs, as well as a Xilinx Zynq FPGA
system that includes a hard-wired ARM processor.

5.1 MapleHoney Reconfigurable Computing Cluster

As our main demonstration platform, we have assembled a hybrid cluster of four FPGAs and
four PCs named MapleHoney. A block diagram of the system is shown in Figure 5.1. Each
of the PCs hold an Intel Core i5-4440 quadcore CPU with 6MB shared L2 cache, running at
3.1GHz, and 8GB of DDR3-RAM. The PCs are fully connected through a Gigabit Ethernet
switch.

5.1.1 BEEcube BEE4

The central component is a BEEcube BEE4 multi-FPGA board. It holds four Xilinx Virtex-
6-LX550T FPGAs, each with the equivalent of more than 500000 logic cells and 22MBits of
on-chip SRAM blocks. The four FPGAs are connected in a ring topology by differential double-
data rate I/O lines providing a 400MB/s communication bandwidth per direction between two
communicating FPGAs. Each FPGA is connected to two separate memory channels with one
8GB DDR3-1066 DRAM module each. Every FPGA is paired through a 4-lane Generation 1 PCIe connection with one of the PCs.

Figure 5.2 illustrates the internal structure of a single FPGA. A fully connected NetIf network based on 32-bit-wide Fast Simplex Links connects all GASNet nodes, the PCIe host interface and the on-board connections to the two neighbouring FPGAs in the ring topology. Each of the two memory controllers can be connected to one, two or four hardware GASNet nodes, resulting in two to eight nodes overall per FPGA. As previously introduced, a THGASNet FPGA node is always composed of a GAScore and a computation component that can either be a MicroBlaze processor or a custom application core. The two components share an AXI bus connection to a memory controller; they can both access the memory as masters. GAScore provides the connectivity to NetIf. All on-chip components are pcore-type IP modules as employed by the Xilinx Embedded Development Kit [60] used for our platform.
5.1.2 Platform-specific components

**Split-range memory controller with AXI interfaces**

The DRAM controller provided to us by BEEcube has a so-called *User Interface*: In contrast to a *Native Interface* that still employs separate bank, row and column addressing, DDR memory can be accessed with a single address that is aligned on 64-byte burst boundaries. Only complete 64-byte read and write bursts are supported; this burst data can be read and written through a 256-bit data port clocked at 200MHz.

To maintain a portable memory interface with flexible configuration options for our GASNet nodes, we have however chosen the AXI bus \(^1\). We were also intent on keeping the variability of the AXI bus, which enables transfers of variable size with no consideration to DRAM burst

---

\(^1\)While Xilinx provides a memory interface generator for various options including an AXI bus, we were not able to use this tool due to the fact that BEEcube designed the BEE4 PCB outside the pinout constraints enforced by our version of the generator tool.
alignment. Furthermore, as we intended to allow more than two GASNet nodes per FPGA, two
memory interfaces would also not have been sufficient and some kind of sharing arrangement
was necessary.

We have tackled these challenges by implementing an additional memory access layer that
translates random burst accesses from up to four AXI buses to the burst-aligned accesses
provided by the User Interface. Our axi_split_mpmc IP core shown in Figure 5.3 has a separate
AXI slave section for each connected bus, including a small local write-through cache that can
buffer a complete burst.

When an AXI read request comes in through one of the connected buses, a read request
to the memory controller is queued and executed as soon as the AXI slave section gets access.
The returned burst of data is stored in the local buffer and the requested part of the burst is
returned over the AXI bus at the available speed. In the meantime, memory requests from the
other AXI sections can be served. Unfortunately, BEEcube has decided not to implement byte-
aligned write masking (DM signals at the DRAM module interface) in the BEE4’s PCB and
memory controller design. As a consequence, any write access that is not a complete, aligned
and unmasked 64-byte burst needs to be implemented as a read-modify-write burst, significantly
and unnecessarily hampering performance for applications that need to write smaller subsets of
a burst. Those AXI write accesses are therefore stalled until the appropriate complete burst of
data has been read into the section’s burst cache. Only then, the write access can modify the
intended bytes, the complete burst can be sent back to the DRAM and the AXI write access
can be signaled as complete.

Our design differs from a typical multi-port memory controller because it does not share
the same memory location between multiple AXI ports. Instead the existing physical range of
memory is split up into equal-sized separate sections for separate AXI buses. This crude “vir-
tualization” of the available memory is necessary because in our programming model, separate
GASNet nodes do not share physical memory. When memory is declared as shared, this instead
means that remote accesses to that memory are possible through the GASNet networking in-
frastucture. As such, it is possible that a Long Active Message induces a GAScore to read
through AXI from one section of a DRAM module, the message is transmitted through NetIfs
to another GAScore, and that GAScore might be connected to the same controller and write
Figure 5.3: Split-range memory controller with separate AXI interfaces for up to four GASNet nodes

the data back into a different part of the same module. The copy itself is absolutely necessary to keep up the logical division of GASNet nodes, however a very well-tuned (and much more complex) future version of our infrastructure might optimize the process and therefore the message latency, for example by letting the same GAScore write the data back to memory. It is, however, important to note that the whole logical partitioning is only necessary because there are not sufficient external memory resources for more than two GASNet nodes in our system. We hope that a future system might offer more physically separate channels, for example in a configuration with *Hybrid Memory Cubes* [61] or *High Bandwidth Memory* [62].

Arbitration between the AXI ports is handled by round-robin. Alternating between AXI sections does not incur a penalty in the form of constant row address switches, as the different AXI port’s address ranges are split up through bank addressing (a common DDR3 memory has eight banks; the provided BEE4 memory controller can be configured to use the most significant
address bits for bank selection).

Table 5.1 lists the resource utilization for both the provided memory controller as well as the different port configurations for our split-range memory controller. As can be observed, the resource usage of the actual memory controller is much larger with the exception of the Block-RAMs used for burst buffering in our AXI interfaces. This buffering could be implemented with LUTRAM if preferable, or alternatively the BlockRAMs could be utilized better by expanding each port’s buffer into a full cache for more than one burst.

Table 5.1: Post-map resource utilization of memory controller and multi-port AXI interface

<table>
<thead>
<tr>
<th>Core/configuration</th>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR3 MIG (BEEcube)</td>
<td>8237 (1.20%)</td>
<td>6053 (1.76%)</td>
<td>0 (0.00%)</td>
</tr>
<tr>
<td>Split-range controller, 1 AXI port</td>
<td>114 (0.02%)</td>
<td>190 (0.06%)</td>
<td>8 (1.27%)</td>
</tr>
<tr>
<td>Split-range controller, 2 AXI ports</td>
<td>221 (0.03%)</td>
<td>592 (0.17%)</td>
<td>16 (2.53%)</td>
</tr>
<tr>
<td>Split-range controller, 4 AXI ports</td>
<td>439 (0.06%)</td>
<td>1308 (0.38%)</td>
<td>32 (5.06%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

It is important to emphasize that the split-range memory controller is not an addition that we envision for every TheGASNet platform, but rather an accommodation of the existing properties of the BEE4 system. GAScores and THeGASNet nodes can utilize any memory controller that offers one or more AXI slave interfaces. The ideal platform offers a single, independent memory channel for each GASNet node. If shared memory is used, it is preferable in terms of efficiency to use a single GASNet node and parallelize the work inside the node’s application core according to a shared memory programming model, as would be recommended in a software system. We elaborate further on our ideal vision of shared and distributed memory use in Section 9.1.2.

**PCle interface**

The core PCIe functionality is implemented by a proprietary netlist designed by Xillybus [63]. It has been provided free of charge to us for research purposes. Unfortunately, the provided IP only supports communication on one of the four PCIe lanes. Xillybus PCIe is modeled as a bi-directional data buffer. On the software side, a Linux character device is provided. Buffers of data can be written to or read from the device through POSIX `read()` and `write()`
calls. Threads can be suspended by calling `read()` until data is provided by the device. On the hardware side, Xillybus provides FIFO-type interfaces to read from and write to the host. They are synchronized to the PCIe bus timing and clocked at 125MHz. We use a 4KByte asynchronous FIFO on both the read and write channels to synchronize to our internal 100MHz system clock. The whole component is encapsulated as a `pcore` that connects to its own NetIf; on the FPGA, the PCIe interface therefore has its own NetIf address, however, as a transport medium it does not have its own GASNet node ID. Table 5.2 shows the resources required in addition to the FPGA’s hardened PCIe core.

Table 5.2: Post-map resource utilization for PCIe core with NetIf interface

<table>
<thead>
<tr>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>3428 (0.50%)</td>
<td>3327 (0.97%)</td>
<td>13 (2.06%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

**FPGA-to-FPGA interfaces**

The BEE4 provides 26 uni-directional, differential-signal, double data rate communication lines per direction for each FPGA-to-FPGA ring connection. We are using these lines to provide an off-chip extension to the 32-bit FSL FIFO channels used between NetIfs on-chip. Through the double data rate support, the 26 lines at 200MT/s are multiplexed/demultiplexed to 52 signals at the internal system clock of 100MHz. These 52 lines are used for the 32 data lines, 7 parity signals to provide SECDED (single error correction, double error detection) Hamming coding per data word, and triple-redundant flow control signals. The complete functionality for a 32-bit receive and a 32-bit transmit channel is again packaged as a `pcore` and interfaces to its own NetIf. Each FPGA has two of these interfaces for “ring-up” and “ring-down” communication. Table 5.3 illustrates that the interfaces require very few reconfigurable resources.

Table 5.3: Post-map resource utilization for `fsl2ic` off-chip communication link

<table>
<thead>
<tr>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 (0.00%)</td>
<td>293 (0.09%)</td>
<td>0 (0.00%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)
5.1.3 Synthetic communication benchmarks for the MapleHoney platform

While our work described in this thesis is not quantitative in character, we are interested in the process of characterizing our platform’s performance with THeGASNet components. When implementing and performance-tuning applications on future productive use heterogeneous PGAS systems, it is important to be able to profile application performance and isolate which system properties influence or limit performance. It is also helpful to understand how the convenience provided by our components is contrasted by introducing performance overhead.\(^2\)

In this section, we practically demonstrate the instruments that allows us to profile performance by extracting latency and bandwidth measurements for the various communication mechanisms offered by THeGASNet. We are taking these measurements on our main demonstration platform, the MapleHoney cluster.

All measurements involving FPGA nodes were done by programming the *Programmable Active Message Sequencer* (PAMS) in each node. In contrast to PC software and operating systems, PAMS and other debugging tools like on-chip logic analyzers allows to do cycle-accurate, deterministic realtime measurements of events on the FPGA or, in our case, on several synchronized FPGAs. At the same time, PAMS gives us the flexibility of software code, enabling agile testing and modifications with a quick turnaround time. An example of generating PAMS code for one of our measurements is attached in Appendix D.2.

FPGA-FPGA communication

Using our messaging infrastructure to connect FPGA components with each other introduces an overhead in comparison to straightforward, point-to-point connecting signals. Each communication request from one FPGA GASNet node to another is processed in the PAMS, the GAScore and the NetIf infrastructure in several sequential, pipelined steps leading to latency cycles. Our default system frequency is 100MHz, so that each cycle of latency translates to 10ns of actual delay.

By using the PAMS’ timer capability, we have measured the cycles to send a *Short*-type Active Message from one node to another ("one-way", Table 5.4) as well as the time for sending

\(^2\)For a discussion of THeGASNet productivity benefits and associated overhead, see Chapter 7.5.
messages back and forth between two nodes (“two-way”, Table 5.5). Because of the four-FPGA ring topology of our system, this was measured for messages inside a single FPGA as well as between two neighbouring FPGAs and two FPGAs not directly connected. The maximum additional distance involved is two off-chip communication hops; as all FPGAs use the same clock and FPGA-to-FPGA interfaces are synchronous with our on-chip clock, we get an exact, reproducible number of latency cycles for off-chip communications.

### Table 5.4: Latency for a one-way Short Active Message

<table>
<thead>
<tr>
<th>FPGA ring distance</th>
<th>cycles</th>
<th>µsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>0.20</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>0.29</td>
</tr>
<tr>
<td>2</td>
<td>38</td>
<td>0.38</td>
</tr>
</tbody>
</table>

### Table 5.5: Latency for a two-way Short Active Message bounce

<table>
<thead>
<tr>
<th>FPGA ring distance</th>
<th>cycles</th>
<th>µsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41</td>
<td>0.41</td>
</tr>
<tr>
<td>1</td>
<td>59</td>
<td>0.59</td>
</tr>
<tr>
<td>2</td>
<td>77</td>
<td>0.77</td>
</tr>
</tbody>
</table>

We are approaching and implementing distributed communications from a software perspective. As such, an area of special impact resulting from the communication latencies shown above is synchronization between system nodes. In Table 5.6, we indicate the latency involved in barrier synchronization, where all nodes are sending a signal to one control node; as soon as all the nodes’ signals have arrived, the control node sends a completion signal back to each node. We have measured the barrier latency for four nodes on a single FPGA, as well as for 16 nodes across all four FPGAs. In each case, all the nodes initiate their requests at the same pre-set time. As a variation on the simple barrier mechanism, we have also measured a distributed barrier, in which the nodes in each FPGA first report to a collecting node in the same FPGA, and only those intermediate nodes communicate with the overall control node. Even in a comparably simple system of 16 nodes, we are saving 24% of latency because congestion on the off-chip communication channels and at the control node is minimized. Similar patterns are essential to scalability by improving synchronization and collectives performance in larger
systems.

Table 5.6: Latency to complete a barrier

<table>
<thead>
<tr>
<th></th>
<th>cycles</th>
<th>µsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 FPGA, 4 nodes</td>
<td>61</td>
<td>0.61</td>
</tr>
<tr>
<td>4 FPGAs, 16 nodes</td>
<td>190</td>
<td>1.90</td>
</tr>
<tr>
<td>4 FPGAs, 16 nodes, distributed barrier</td>
<td>145</td>
<td>1.45</td>
</tr>
</tbody>
</table>

In Table 5.7, we document latencies to copy data between two nodes’ memories with a Long-type Active Message, starting with a minimum transfer of a single 32-bit word up to the transfer of a single packet of maximum size (8KByte). The latencies now include both GASNet-induced latencies (Long-type packet headers are longer than for Short messages) as well as the memories’ read and write latencies (to ensure write consistency, GAScore only triggers a handler function when a write has completed). We have provided minimum, maximum and average latencies, as the memory access latencies have occasional variations, likely due to DRAM refresh. All measurements have been taken on a single FPGA, though with separate memories; off-chip connections will add the same latencies as for Short-type messages.

Table 5.7: Latency for memory copies (Long Active Messages) inside a single FPGA

<table>
<thead>
<tr>
<th>Transfer size</th>
<th>cycles(min)</th>
<th>cycles(max)</th>
<th>cycles(avg)</th>
<th>µsecs(avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>76</td>
<td>91</td>
<td>76.7</td>
<td>0.77</td>
</tr>
<tr>
<td>64 bytes</td>
<td>95</td>
<td>108</td>
<td>95.2</td>
<td>0.95</td>
</tr>
<tr>
<td>256 bytes</td>
<td>260</td>
<td>289</td>
<td>262.5</td>
<td>2.63</td>
</tr>
<tr>
<td>1 KByte</td>
<td>920</td>
<td>950</td>
<td>931.0</td>
<td>9.31</td>
</tr>
<tr>
<td>8 KBytes</td>
<td>7144</td>
<td>7180</td>
<td>7159.9</td>
<td>71.60</td>
</tr>
</tbody>
</table>

Table 5.8 indicates the transfer bandwidth for sustained remote memory copies inside a single FPGA. As the NetIf network bandwidth for 32-bit channels at 100MHz should be close to 400MB/s, the smaller numbers are due to memory access latencies. These could likely be improved with a memory controller that is optimized for continuous burst transfer. We can only detect a small difference between transfers from and to the same memory controller and module and between to two different controllers respectively modules (as we explained in Section 5.1.2, two or four nodes can share the same memory module, separated by banks; in this design, there were only two nodes per module/controller).
Table 5.8: Sustained transfer bandwidth (Long Active Messages) inside an FPGA

<table>
<thead>
<tr>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64MByte transfer, node 0 to 1 (shared DRAM module)</td>
</tr>
<tr>
<td>64MByte transfer, node 0 to 3 (different DRAM modules)</td>
</tr>
</tbody>
</table>

To simulate real application behaviour, we have also tested sustained communication between multiple nodes as documented in Table 5.9. A single node will constantly send large packets to other nodes according to a pseudorandom pattern. As can be observed, the bandwidth inside an FPGA is higher because all NetIfs share a fully connected topology, while there is only one duplex NetIf channel each between two FPGAs.

Table 5.9: “Flooding” - sustained all-to-all communication

<table>
<thead>
<tr>
<th>System size</th>
<th>Average send/receive bandwidth per node (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 FPGA, 4 nodes</td>
<td>128.61</td>
</tr>
<tr>
<td>4 FPGAs, 16 nodes</td>
<td>92.57</td>
</tr>
</tbody>
</table>

**PC-FPGA communication**

We have also examined the communication between PC and FPGA nodes, which happens via PCIe in our MapleHoney system. As the PCs and the Scientific Linux operating system we are using lack the hard realtime capabilities available through our FPGA hardware, measurements are averaged by nature to hide the significant, non-deterministic latencies that occur because of context switches and varying I/O allocation. Table 5.10 shows the average latency of a two-way Active Message between a software node and an FPGA node, measured by sending Short messages back and forth a 1000 times; there is no way for us to find out if one-way messages take different times depending on the direction.

Table 5.10: Latency for a two-way Short Active Message communication

<table>
<thead>
<tr>
<th>µsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average of a 1000</td>
</tr>
</tbody>
</table>

Table 5.11 shows the bandwidth measured when initiating a bulk remote memory transfer between a PC node and a hardware node or vice versa. Since the value for FPGA-to-PC
bandwidth is nearly identical to the one measured for FPGA-to-FPGA communication in 5.8, we suspect the memory read bandwidth on the FPGA as the bottleneck. Our single-lane, PCIe Generation 1 connection should optimally sustain 250MB/s.

<table>
<thead>
<tr>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64MByte transfer, PC to FPGA</td>
</tr>
<tr>
<td>64MByte transfer, FPGA to PC</td>
</tr>
</tbody>
</table>

### PC-PC communication

Finally, we have explored communication between THeGASNet software nodes on our Maple-Honey PCs. Table 5.12 shows the latency for two-way messages between software nodes on the same PC as well as between two PCs across TCP/IP and Gigabit Ethernet. Unsurprisingly, communication on the same processor system is significantly faster than across the network. To estimate bandwidth, we have again set up a pseudorandom all-to-all communication pattern between several software nodes. As for short messages, communication on a single-processor system is much more efficient, especially since we use a shared-memory threading model and avoid double-copying when messaging between local GASNet nodes.

<table>
<thead>
<tr>
<th>µsecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-to-thread, average of a 100</td>
</tr>
<tr>
<td>Across Ethernet, average of a 100</td>
</tr>
</tbody>
</table>

It should be pointed out that the use of both TCP/IP and 1-Gigabit-Ethernet is far below the technological state-of-the-art and networking could be tremendously improved through faster networks and leaner protocol layers.
Table 5.13: “Flooding” - sustained all-to-all communication

<table>
<thead>
<tr>
<th>System size</th>
<th>Average send/receive bandwidth per node (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PC, 4 thread nodes</td>
<td>1520.6</td>
</tr>
<tr>
<td>4 PCs, 16 thread nodes</td>
<td>29.20</td>
</tr>
<tr>
<td>4 PCs, 16 thread nodes, no local communication</td>
<td>22.42</td>
</tr>
</tbody>
</table>

5.2 Digilent Zedboard

*NOTE:* Zynq-specific software and hardware modifications and designs have all been implemented by M.A.Sc. Sanket Pandit and are part of his original research [50]. We are describing these components here only to demonstrate the multi-platform capabilities and portability of the THeGASNet library and GAScore components, not to claim them as our own accomplishments.

As our implementation target for ARM-based THeGASNet applications and designs we have chosen the popular *Digilent Zedboard* [64], which is built around a Xilinx Zynq Z7020 reconfigurable system-on-chip. Figure 5.4 shows a system overview including an example THeGASNet setup.

The hardwired Processing System (PS) part of the Zynq chip includes dual-core ARM Cortex-A9 processors, a 1Gb Ethernet MAC with Direct Memory Access and the DDR3 memory controller. On the Zedboard, the hardwired memory controller connects through a 32-bit-wide port to 512MBBytes of DDR3-1066 DRAM. In the Programmable Logic fabric (PL), components can directly access the memory controller through burst-enabled AXI master ports (AXI_HP).

In the illustrated setup, the ARM processor cores are each running a THeGASNet software node. The programmable fabric includes two THeGASNet FPGA nodes, which are each composed of a GAScore plus either a MicroBlaze processor or a custom hardware core. GASNet communication between software and hardware nodes is accomplished through a Zynq-specific NetIf DMA core, which has been developed as an equivalent to the PCIe interface on the Maple-Honey cluster. Data transfers into or out of the programmable fabric can be accomplished through POSIX `read()` or `write()` calls that trigger DMA transfers from software memory into the NetIf interconnect or back. To the software and hardware nodes, this Zynq-specific solution is of course opaque and hidden behind the THeGASNet APIs. The Zedboard implementation
suffers from two major bottlenecks. The only available external memory is shared between the processing system and the programmable fabric; at 512MBytes for the Linux kernel, software applications as well as hardware cores it is also not exactly plentiful. Other Zynq platforms exist that provide the programmable fabric with its own memory, which is also a partition that is much better served by our programming model. However, we only had access to Zedboards when starting to port THeGASNet to Zynq. Secondly, the only way for the FPGA components to communicate with other FPGA components is through the software-controlled Gigabit Ethernet. Sending an Active Message from a hardware node to a hardware node on another Zedboard involves at minimum one NetIF DMA transfer and one Gigabit Ethernet DMA transfer on each of the boards. This optimistically assumes no copying in the TCP/IP stack.

Note: Synthetic communication measurements for the Zedboard similar to the ones in Section 5.1.3 can be found in Sanket Pandit’s thesis [50]. They include on-chip measurements as well as cluster performance of multiple Zedboards.
5.3 Portability to other platforms

While we are currently only using two different FPGA platforms, the BEE4 and the Zedboard, we are not strictly bound to these specific targets. In the same way that a software API can abstract away varying computer hardware on lower levels, our GAScore and AXI bus interfaces abstract away specific on- and off-chip networks as well as specific memory infrastructures. All of our computation cores (see applications in Chapter 6) use the same Fast Simplex Link interfaces to the GAScore and an AXI burst master interface to memory. As AXI can bridge both different data widths and asynchronous clock speeds, computation cores can even be optimally clocked and equipped with the best data width for the application. Moreover, all components for off-chip connectivity like the PCIe core and the BEE4 inter-FPGA links use NetIf interfaces into the fabric. These interface abstractions on both sides mean that any computing component can be used on different FPGA systems that use varying memory controllers with AXI interfaces (the Xilinx-Spartan-6 series, for example, has a hardwired memory controller with an AXI interface), and any off-chip interface can be used if there’s a NetIf connection. When changing to a different chip or a different board, we only need to make sure that the different components available conform either to the AXI or NetIf interfaces. For example, the PCIe-based system was originally implemented and tested on a Virtex-6-LX240T chip, essentially a smaller version of our current LX550T chips. Moving our cores to the Zynq programmable fabric was possible as soon as a NetIf interface to the ARM system existed. We are currently bound to Xilinx through some vendor-specific primitives and IP in our cores, however, these components are available at a minimum for the Virtex 5 through 7 and Zynq families and equivalent functionality for other vendors is available.
Chapter 6

Application use cases

In this chapter, we are going to practically demonstrate how to use our GASNet-based design flow with three different application examples, each of which displays a different communication pattern. In all cases, the application is first implemented completely in software and then critical parts of the application are isolated and moved to hardware.

As a reminder, Figure 3.2 showed what a software-only PGAS stack utilizing the original GASNet looks like. The Core and Extended APIs abstract the network and provide one-sided communications functionality to either a language-specific PGAS runtime (e.g. UPC, Chapel) or a PGAS application library (e.g. Global Arrays, SHMEM).

Our currently existing stack for the three heterogeneous demonstration applications can be seen in Figure 6.1. As the stacks differ significantly for FPGA hardware and software, we only show the PC/FPGA networking fabric as a common component (we have also left out the stack

![Figure 6.1: Heterogeneous test application stack using THeGASNet](image_url)
seen by a MicroBlaze software node, which is a mix of the two). The important aspect from the application programmer’s perspective is that THeGASNet will completely abstract these different networking infrastructures away.

On the PC side (left), we use our own THeGASNet software implementation of the GASNet Core API. On the FPGA side, the Core API is provided as a hardware interface by GAScore. The software implementation is enhanced by integrating functionality for unlimited-size Active Message transfers, which is arguably a substitute for the Extended API Put() mechanism. We are not explicitly using an Extended API, but in addition to the unlimited-size messages, our C applications use functions for synchronization barriers and the subgrouping of nodes (the latter only for the BFS and FFT/encryption applications). As we are using standard C for our applications, there is no specific PGAS runtime; neither is there a PGAS library - the applications use the THeGASNet API directly.

On the FPGA side (right), the Core API provided by GAScore is directly connected to the PAMS, which implements barriers and unlimited-size transfers. The handwritten code in the PAMS can therefore be seen as equivalent to an Extended API layer. The actual computing applications are hand-written in VHDL and use the interface to the PAMS for all remote communications.

### 6.1 Jacobi Relaxation of Electrostatic Differential Equation

The Jacobi method is an iterative numerical method to solve partial differential equations on a discrete spatial grid. For our application, we are taking an example from electrostatics: The relationship between the electric potential \( V(x, y) \) and the distribution of static charge \( \rho(x, y) \) in a two-dimensional area is defined by the Poisson equation:

\[
\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) V(x, y) = -\frac{\rho(x, y)}{\epsilon_0}
\]

When discretizing this equation on a grid of pitch \( h \) and normalizing \( \rho \) by \( \epsilon_0 \), the potential at each grid location is defined as a combination of its local charge and the potential of its vertical
and horizontal neighbours:

\[ V_{x,y} = \frac{V_{(x-1),y} + V_{x,(y-1)} + V_{x,(y+1)} + V_{(x+1),y} + h^2 \rho_{x,y}}{4} \]

This relationship is useless as long as none of the grid points are known. Jacobi’s method, however, allows the calculation of values at all locations iteratively by starting from random values for \( V(x,y) \) and re-calculating each point’s value in iteration \( t \) from neighbouring values from the previous iteration \( t - 1 \) until the value distribution converges:

\[ V_{t+1,x,y} = \frac{V_{t,(x-1),y} + V_{t,x,(y-1)} + V_{t,x,(y+1)} + V_{t,(x+1),y} + h^2 \rho_{x,y}}{4} \]

As a computation pattern, the Jacobi iteration can be understood as a stencil operation (see Figure 6.2), which re-computes a grid point or cell from neighbouring information. This implies high locality of the problem, making it relatively straightforward to parallelize. The complete grid area can be partitioned into smaller segments that different computation nodes can work on (see Figure 6.3). Individual nodes can recalculate a new iteration for each of its cells without involving other nodes. However, the problem cannot be classified as embarrassingly parallel since between iterations, neighbouring cells from other nodes must be communicated to give the cells at the edge of each partition all necessary input values (see Figure 6.4). Each full Jacobi iteration must therefore be preceded by a retrieval of edge information from neighbouring nodes.

---

1 Convergence is dependent on the properties of the linear equation system that describes the potential-charge relationship in the grid; suffice to say here that the convergence conditions are fulfilled for the applicable Poisson equation.
Figure 6.3: Node partitioning

Figure 6.4: Node-to-node communication between iterations
partitions. However, the problem scales well as, for the 2-D problem, a linear increase of the problem size only leads to a square-root increase in communication.

The grid points at the edge of the problem area do not have neighbours to retrieve information from. However, we will still need to supply them with neighbouring values as these will serve as boundary conditions for the differential problem. We are initializing all boundary cells to a constant value of 0 (Dirichlet conditions), which in our concrete physical problem would equate to a grounded conductive frame or shield around the grid area.

Finally, the iterations of neighbour communication and subsequent communication need to be repeated until the data grid converges sufficiently towards a steady state. An appropriate metric that qualifies convergence needs to be picked. One possible example is that the average quadratic error between two iterations of cell values declines under a pre-defined threshold. When this threshold is reached, the computation is considered finished.

6.1.1 Software-only implementation

In our initial software implementation, we are using 16 THeGASNet software nodes, one node each per processor core in the MapleHoney constellation of four quad-core host PCs. Starting with a grid size of 32768x32768 cells, we end up partitioning our nodes into a 4x4 grid with each node calculating 8192x8192 cells. Using 32-bit integer values, one node partition requires 256 MBytes. However, as our iterations require the cells for iterations $t$ and $t + 1$ both, we need to allocate two sets of cells that we will use to read from one and write to the other in an alternating fashion. The newly written block on each iteration becomes the read source for the next iteration, correspondingly the read source of this iteration becomes the destination to write to on the next iteration. Furthermore, we need to allocate space to store the edge cell information from neighbouring nodes, so that we end up requiring 512.25 MByte per node. A high-level description of our implementation is shown in Algorithm 1: The essential steps are the initialization, the communication of edges, the computation of cells and finally the decision whether to keep iterating. In this last step, one of the nodes, arbitrarily decided to be number 0, functions as a master node that accumulates state data from all nodes, makes a decision on convergence and signals this decision to all others. Aside from that, all nodes perform a near-identical workload (nodes on the edge of the data grid do not communicate to as many
Algorithm 1 Jacobi relaxation on 16 software nodes

1: procedure THEGASNet_JACOBI_RELAXATION
2: Compute partition size
3: Allocate blocks IN, OUT
4: Initialize blocks to 0
5: repeat ▷ iteration loop
6: Send edge cells from IN to neighbour nodes ▷ communication
7: wait for all edge data to arrive
8: for all local cells in block OUT do ▷ computation
9: Read neighbouring cells from block IN
10: Combine neighbours, local charge
11: Write cell to OUT
12: Square difference between cell in IN and OUT, accumulate
13: Swap IN and OUT
14: if Node==0 then ▷ convergence check
15: Accumulate sums-of-squares from all nodes
16: if Convergence threshold has been reached then
17: Signal nodes to stop
18: else
19: Signal nodes to continue
20: else
21: Send sum of squares to Node 0
22: wait for Node 0 reply
23: until Node 0 signals stop
The detailed distributed application works as expected on the MapleHoney PCs. Furthermore, the application can be identically compiled for the 16 MicroBlaze processors that can be used in a 4-per-FPGA configuration on the BEE4 board. The performance in this case is not competitive with the x86-64 implementations because of the much slower performance of the MicroBlaze processor.

6.1.2 Isolation of accelerator code and hardware modeling

In the described algorithm, it is the compute-intensive code that we would like to migrate into hardware accelerators. This entails the complete inner loop that calculates new cell values for each iteration and accumulates the values for the convergence criterion. Furthermore, the communication with other nodes should be initiated directly from the hardware engine with the help of GAScore. On the other hand, less compute-intensive tasks that involve control flow, decision-making and computation of system parameters are likely better kept in a software node.

Before actually moving to hardware cores, we are using the flexibility GASNet gives us to model the future hardware behavior in software. This modeling does not necessarily involve the exact way how a hardware core will accomplish its task (although software modeling can be useful for that purpose), but how it is going to interact with the other nodes.

As an initial design, we will prepare for using 16 hardware GASNet nodes equivalent to the 16 software nodes used for computation. However, since we still want a software node in our target design, we will now extend the number of nodes to 17 and partition our current algorithm into two different versions as seen in Algorithms 2 and 3. To a large extent, the two algorithms are just separated halves of the original algorithm. As intended, 16 computation nodes do all the “hard” (but regular) work, while the master node focuses on the centralized decision-making when to continue or stop the communication. Additionally, the master node is now the only node that processes (command-line) parameters that specify the problem size and the number of available GASNet nodes, decides on partition sizes and sends those parameters to the computation nodes to use for their operation.

The described setup of 17 nodes works equally well running exclusively on the MapleHoney
Algorithm 2  Jacobi relaxation - code for master node (0)

1: procedure THEGASNet_JACOBI_RELAXATION_MASTERNODE
2:   Compute partition size
3:   Distribute partition parameters to other nodes
4:   repeat                     \(\triangleright\) iteration loop
5:     Accumulate sums-of-squares from all nodes \(\triangleright\) convergence check
6:     if Convergence threshold has been reached then
7:         Signal nodes to stop
8:     else
9:         Signal nodes to continue
10:    until Convergence threshold has been reached

Algorithm 3  Jacobi relaxation - code for computation nodes (1-16)

1: procedure THEGASNet_JACOBI_RELAXATION_COMPUTATIONNODE
2:   Initialize shared memory to 0
3:   Receive partition parameters from Node 0
4:   repeat                     \(\triangleright\) iteration loop
5:     Send edge cells from IN to neighbour nodes \(\triangleright\) communication
6:     wait for all edge data to arrive
7:     for all local cells in block OUT do \(\triangleright\) computation
8:         Read neighbouring cells from block IN
9:         Combine neighbours, local charge
10:        Write cell to OUT
11:        Square difference between cell in IN and OUT, accumulate
12:    Swap IN and OUT
13:    Send sum of squares to Node 0
14:    wait for Node 0 reply
15:    until Node 0 signals stop
PCs or as a mixed configuration of a PC master node and 16 MicroBlaze computation nodes, the latter of course with the previously mentioned performance loss. With the intended task for custom hardware cores now clearly isolated, we can start designing application-specific hardware.

6.1.3 Jacobi hardware core

The existing GAScore and memory bus framework that is in place for both MicroBlazes and custom cores pre-defines the following port layout:

1. An AXI bus master port, which can burst-read and -write data from the main memory shared with the GAScore. The Jacobi core will read data from one buffer in main memory, and write it back to the second buffer.

2. Duplex FSL connections to the GAScore to communicate Active Message requests and receive handler information about incoming messages. As we are inserting a PAMS instance for the communication handling, the required port reduces to simple control signals to synchronize operation with the PAMS and a data port to receive parameters and return a value for the convergence criterion.

The actual implementation of the core is quite straightforward given the computation pattern. The core reads three rows of input into local SRAM buffers with minimal access latency. At this point, the three row buffers can be read out in lockstep, yielding the stencil information for a new row of results to be written back to the main memory. Concurrently to the write-back, a sum of squared differences is accumulated for the convergence criterion. While the new row is computed and written, a fourth row buffer can already read the next row from main memory. After the first row of results has been written, the second and third row, together with the newly read fourth row, can be used as input for the stencil computation of the second row of results. The very first input row can at this point be discarded, and the buffer can be refilled with the fifth row of input data. This process repeats until all the newly computed rows have been written.

As SRAM storage on the FPGA is scarce, a single stencil core might not have enough storage capacity for four complete rows. Therefore, when the row size is too long for the available buffer
depth, the core will read part of each of the first four rows as described before, and so on. The
node’s data is therefore actually processed in strips or columns as wide as one row buffer is
deep.

Table 6.1 shows the resources required for a single Jacobi core. Resource use is dominated
by the on-chip BlockRAMs for the line buffering. Each core uses four DSP48 slices to implement
the multiplication that determines the squared error metric between iterations.

<table>
<thead>
<tr>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
<th>DSP48</th>
</tr>
</thead>
<tbody>
<tr>
<td>3310 (0.48%)</td>
<td>4448 (1.29%)</td>
<td>65 (10.28%)</td>
<td>4 (0.5%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

6.2 Breadth-First Search (Graph500)

The TOP500 list of supercomputers is accumulated by ranking the systems based on their
\textit{FLOPS} (floating point operations per second) performance when running \textit{HPLinpack} \cite{65}, a
software for distributed solving of systems of linear equations through \textit{LU decomposition}.

Led by the observation that linear equation solvers only represent a subset of existing large-
scale computing applications, Murphy et al. \cite{66} have suggested that a ranking of computing
performance for graph algorithms would be beneficial. Accordingly, this would help assess per-
formance on problem classes related to data analytics. The specific suggestion for a GRAPH500
list is a \textit{Breadth-First Search} (BFS) on an undirected, cyclic graph. A detailed description of
the benchmark can be found under \cite{67}.

The basic operation is defined in Algorithm 4 and illustrated in Figure 6.5. Starting from a
specified vertex \(s\), the algorithm traverses all its edges to find all other vertices that \(s\) is directly
connected to. These vertices are marked to be processed on the next iteration and assigned \(s\) as
their parent. On each iteration, the newly found vertices from the last iteration are processed
by traversing each vertices’ edges and marking the newly found neighbouring vertices (i.e. the
ones that do not have a parent assigned yet). The algorithm is finished when no new parentless
vertices have been found during an iteration. The problem definition does not require a specific
order in which to iterate over the vertices found in the last iteration or an order in which to
Figure 6.5: Breadth-first search example: Undiscovered vertices in white, newly discovered vertices in light grey, currently parsed vertices in middle grey, vertices from previous iterations in dark grey.
Algorithm 4 Breadth-first search

1: **procedure** BFS(Graph G, Vertex s)
2:     **for** each Vertex v in G **do**
3:         v.parent = NIL
4:     s.parent = s  \( \triangleright \) Vertices to parse
5:     Queue Q = \{s\}  \( \triangleright \) Newfound vertices
6:     Queue F = {}  \( \triangleright \) Newfound vertices
7: **repeat**
8:     while Q\( \neq \)\{} **do**
9:         **for** each Vertex v in Q **do**
10:             Q.dequeue(v)
11:             **for** each Edge e connected to v **do**
12:                 Vertex vn = e.otherEnd
13:                 if vn.parent = NIL **then**
14:                     vn.parent = v
15:                     F.enqueue(vn)
16:             Swap(Q,F)
17: **until** Q = {}  \( \triangleright \) No newfound vertices to parse

traverse a vertex’ edges. As such, the algorithm can be parallelized well, but on each parallel run, a specific vertex can be assigned a different parent from the possible group of parents. The result is therefore not completely deterministic; however, the number of traversed edges and newly found vertices on each iteration are, therefore these metrics can be used to compare two runs of the same problem.

The Graph500 benchmark description defines problem sizes, storage requirements, the exact construction algorithm for building a graph based on random numbers and the performance metrics. It provides a MATLAB code example for both the Kronecker graph edge generator [68] and the graph construction. A graph is defined by it’s *scale*, having exactly \(2^\text{scale}\) vertices, and its *edge factor*, the average number of edges generated per vertex\(^2\). Based on these two parameters, the Kronecker generator will generate the graph’s edges in the form of tuples of two vertices that are supposed to be connected. While the *scale* of the graph can be varied based on the benchmarked system’s storage capacity, the *edge factor* is prescribed to be 16 for all sizes. While this results in an average of 32 edges connected to a vertex, the numbers of edges varies widely, leading to a significant fraction each of vertices with no edges and thousands of edges. Independent of the actual graph scale being benchmarked, Graph500 requires a minimum

\(^2\)Note that, because the graph is undirected, each vertex will on average have \(2*\text{edge}\_\text{factor}\) edges to traverse
storage size of 48 bits for each vertex identifier. Most platforms will likely, for reasons of word alignment, reserve 64 bits per vertex; the 16 bits of overhead can be used for marking of vertices with additional information.

The benchmark documentation requires the timing and submission both of the graph construction (called “Kernel 1”) time as well as the actual breadth-first search (called “Kernel 2”) performance, however only BFS performance is used for the ranking. The metric used in the ranking is \textit{TEPS} (traversed edges per second) and is therefore not directly related to graph scale. After constructing a graph once, the breadth-first search needs to be run 64 times from different randomly selected starting vertices, excluding vertices that have no edges or only self-loops. The TEPS performance of the 64 runs is then averaged.

### 6.2.1 Software-only implementation

For any practical implementation of BFS, we must first decide on a data structure to store our graph. The two most common approaches to storing graph data are the \textit{adjacency matrix} and the \textit{adjacency list} representation.

An \textit{adjacency matrix} has as many rows and as many columns as the graph’s number of vertices. Each matrix element represents a value specifying the number of edges between two vertices, one in the column position and one in the row position. For an undirected graph, one diagonal half of a matrix would suffice, as each undirected edge turns up twice in the matrix. However, for a distributed system we would likely split the matrix into slices, losing the full redundancy locally. Let us assume that we do not need to keep the number of edges connecting two vertices, as that information is irrelevant to the BFS (Graph500 is unspecific on this). We would then only need one bit for each matrix element. As an example, a graph of $2^{22}$ vertices (the scale we ended up implementing on a single PC respectively a single FPGA) would require $2^{44}$ matrix elements or 2 TBytes. Of course, with an average of only 32 edges per vertex, the matrix is very sparse and it would be sensible to compress the matrix in some way. However, this would complicate both the graph construction and parsing the adjacency information significantly.

The \textit{adjacency list} approach stores either a linked list or an array of other connected vertices for every single vertex. The linked list is completely flexible in size; the array size for each
vertex could only be fixed after building the complete graph. Assuming the naive linked-list implementation of a 64-bit vertex field and a 64-bit pointer to the next list element, our example of $2^{22}$ vertices would require approximately 2 GBytes of storage ($2^{22}$ vertices times 32 edges per vertex times 16 bytes per edge); the optimally-sized array-solution would require approximately half of that. Both adjacency list solutions are therefore preferable to the naive adjacency matrix implementation.

To minimize both the graph construction effort and the storage requirements, we have chosen an adjacency list implementation that combines arrays with dynamic linking. We have decided to allocate an array of 31 adjacency fields for each vertex statically. This way, the majority of vertex connections can be accessed from a statically computed memory location and in a contiguous memory burst. We call this array of vertex connections a slice. The downside of this approach is wasting memory space on all vertices with less than 31 edges. The 32nd field in each slice holds a link pointer to a dynamically allocated slice with additional connections. To optimize burst access for vertices with many connections, the size of dynamically allocated slices doubles every two links. The size of a slice and the dynamic growth in slice size have been chosen heuristically and have potential for optimization. However, with the currently chosen parameters we were able to keep our storage requirement below the projected size for the naive linked-list adjacency list implementation; this held true for several dozen graph datasets generated by us.

To produce a distributed version of the algorithm, it needs to be determined how both the data and the workload are to be split up. The most straightforward solution for data distribution is to split up the graph data into blocks of equal numbers of vertices and then store the corresponding adjacency lists accordingly. Theoretically it is, of course, possible to keep a whole copy of the graph on each computation node and just distribute the work, but it does rob a distributed system of its advantage of larger possible problem sizes (enabling the investigation of weak scaling).

With the vertex adjacency lists distributed evenly across nodes, part of the workload is implicitly distributed the same way: Finding the neighbours of a vertex involves iterating over its adjacency list, and is therefore done locally at the point of storage. The second essential part of the algorithm involves checking those adjacent vertices’ parentage status, and marking
and enqueuing the heretofore parentless ones for the next search iteration. As this will lead to parsing those newfound vertices’ adjacency lists on the next iteration, we are storing both the parent information for those vertices and the part of the queue that we put them in for the next iteration on the same node. This results in distributing the parent information and the queue by vertex, too. The storage requirement of the parent list is one parent vertex per child vertex. The queue needs to be able to hold all the graph’s vertices (except the starting vertex) for the pathological case of all of them being connected to the initial vertex. For the previous example of a graph size of $2^{22}$ vertices, the parent list and the queue each need 32 MBytes, split up evenly among nodes; however, the BFS as presented in Algorithm 4 uses two queues which are swapped in between operations. Nevertheless, the storage requirements are dominated by the adjacency list data.

The modified BFS algorithm for a single node in distributed form is shown in Algorithm 5. As described, in a single iteration, the locally enqueued vertices’ adjacency lists are parsed. Retrieved edges connecting to locally stored vertices are enqueued in a local queue, all other edges are sent to the appropriate remote nodes’ queues. Edges in the local queue are checked whether they are connected to a parentless vertex. If this is the case, the vertex is handled as in the basic algorithm. The parent (the originating vertex of the edge) is set and the new vertex is enqueued for processing on the next iteration. All nodes need to be finished before any node can start the next iteration, therefore a global barrier is inserted at the end of the iteration.

The bulk of the communication happens when edge information is sent to other nodes to process. Sending every edge (essentially, the two connected vertices, therefore 16 bytes of data) in a unique message would be enormously wasteful; therefore, edges are accumulated in local buffers and then sent as a Long-type Active Message of maximum size to a memory buffer on the remote node. As it is a direct memory copy, the destination buffer’s address offset needs to be known by the sender. Available buffer addresses are distributed to other nodes at the start of the algorithm. When the address has been used as a destination by the Active message, the receiving handler function provides it to the local application, which processes the new edges. After the buffer has been processed, the buffer address is sent back to the sending node in a

---

3For a single processor, the BFS algorithm can be defined with a single queue and without the explicit demarkation of iterations. However, for a distributed version there needs to be a barrier between iterations, otherwise race conditions producing incorrect parent information are possible.
Algorithm 5 Distributed breadth-first search

1: procedure BFS\_DISTRIBUTED(Vertices Glocal, Vertex s)
2: localsize = size(Glocal)
3: Queue Q = \{\}  \hspace{1cm} \triangleright Vertices to parse
4: Queue F = \{\}  \hspace{1cm} \triangleright Newfound vertices
5: Queue RE = \{\}  \hspace{1cm} \triangleright Received Edges
6: if s ∈ Glocal then
7: Q.enqueue(s)
8: s.parent = s
9: for each Vertex v in Glocal do
10: v.parent = NIL
11: repeat
12: while Q ≠ \{\} do
13: for each Vertex v in Q do
14: Q.dequeue(v)
15: for each Edge e connected to v do
16: Vertex vn = e.otherEnd
17: if vn.parent ∈ Glocal then
18: RE.enqueue(e)
19: else
20: EnqueueInRemoteNode(e, vn/localsize) \hspace{1cm} \triangleright Send to correct node
21: while RE ≠ \{\} do
22: for each Edge e in RE do
23: Vertex vn = e.thisEnd
24: if vn.parent = NIL then
25: vn.parent = e.otherEnd
26: F.enqueue(vn)
27: Barrier() \hspace{1cm} \triangleright Synchronize iteration finish
28: Swap(Q,F)
29: until Q = \{\} \hspace{1cm} \triangleright No newfound vertices to parse
Short-type message to re-use as a destination. The address exchange mechanism is comparable
with a credit-based network; however, transmitting a free address is more specific than just
enabling another transmission with credit. The main difference is that buffers can be filled
and emptied in random order. As such, it is more flexible and involves less synchronization
overhead than similar transfers based on message-passing.

Our current solution allocates 256 buffers of 8KBytes (the maximum Active Message size) for
each communication partner. For a system of 16 nodes (our current maximum amount of BFS
nodes), this amounts to 32MBytes in each node, much less than the space for graph information,
but sufficient to not run out of buffers during communication. Allocating individual buffers for
each communication partner will, however, not scale to very large systems, so that a more
localized distribution mechanism, e.g. a butterfly [69], are likely preferable\footnote{Note that with the term \textit{butterfly} we are not referring to the topology of a system’s communication network, which is completely opaque at the GASNet application level, but to a buffer-based routing mechanism for edge information.}.

In discussing the breadth-first search algorithm, defined as “Kernel 2” by Graph500, we have
so far ignored the initial step of graph construction, identified as “Kernel 1”. The Kronecker gen-
erator defined by Graph500 generates a predefined amount of random edges ($2^{\text{scale} \times \text{edge_factor}}$, 
to be precise), essentially tuples of two vertices, which completely define the graph. However, to
be useful, these edge tuples have to be converted into adjacency information. Graph500 allows
the distributed generation and construction of the graph, meaning that each node randomly
generates a fraction of the edges, however each with all possible vertex numbers, not just the
vertex numbers stored locally later (otherwise, each node would produce its own closed graph).
Each generated edge is then distributed to the up to two nodes that store the adjacency infor-
mation of the two vertices connected by the edge. Receiving nodes build their local adjacency
lists based on the edge information. For the distribution of edges, we use the same buffer
mechanism that we also employ in the breadth-first search.

6.2.2 Isolation of accelerator code and hardware modeling

Isolating the performance-critical part of Graph500 is quite straightforward in our implement-
tion. The software is currently composed of two large kernels, the graph construction and
the graph search. The graph construction is only executed once, while the breadth-first search
can be executed many times. The graph information itself is never modified, there is only a new parent list being produced on every BFS run. While the software implementation of the graph construction runs about an order of magnitude slower than a single BFS run, it is not performance-critical in comparison to the accumulated run time of many BFS runs. On a more pragmatic note, only BFS runs are considered in the Graph500 ranking and therefore worth the effort of implementing hardware acceleration.

Preparing the software for a hardware migration means isolating the parts to be run in hardware - in our case Kernel 2, the breadth-first search - into separate GASNet nodes and identifying the communication patterns between the nodes running the two separate parts. Fortunately, the two kernels are executed strictly in sequence, with Kernel 2 using the data structure built completely by Kernel 1. Separating the kernels into different nodes entails three major modifications:

- The actual code being executed is dependent on the GASNet node ID. Whereas in the original code every GASNet node executed a share of both kernels, we are now splitting the number of GASNet nodes in half and let one half execute Kernel 1 and the other half Kernel 2. Figure 6.6 illustrates the process. Such a grouping of nodes and division of responsibilities is supported in the MPI library as so-called communicators, GASNet however does not support a similar mechanism at the Core API level. We have implemented a small helper library to define node subgroups, assign nodes to these groups in a blocked or cyclic manner, and provide continuous index numbering inside these groups. A Build node executes only the graph construction kernel and a BFS node only runs the breadth-first search kernel.

- As the graph is now only constructed in the memory of the Build nodes, the completed data structure needs to be copied into the BFS nodes’ memory. This reflects the identical required step of copying the graph from PC into FPGA memory.

- As an extension to the definition of subgroups, there also need to be distinct barrier functionalities to synchronize all nodes (for example for GASNet startup, the graph copying and GASNet cleanup) or to synchronize only the Build nodes or only the BFS nodes. Furthermore, as in the Jacobi application, one of the nodes needs to be designated as a control node that communicates with everybody independent of subgroup assignment.
Figure 6.6: Host-to-FPGA migration for BFS: Responsibilities for graph construction (Build) and search algorithm (BFS) are first split into different software nodes; the BFS group of nodes can then be substituted by FPGA nodes with the same communication behaviour. A mix of software and hardware nodes with the same function is also possible.

We designate Node 0, a member of the Build node group, for this, as it will stay a software-implemented node. Besides controlling barriers, it will also collect statistics about newfound vertices and traversed edges after each iteration.

As Section 6.4.2 shows, the best software-only performance for our BFS is reached by running one GASNet node per core on our available quad-core PC systems in the MapleHoney cluster. This is based on GASNet nodes that do not split up computation into multiple threads, which is of course possible to do. Multiple threads per application node might lead to better performance, but would require a reassessment of the optimal number of nodes per processor core.

In the non-partitioned version, a single PC runs four GASNet nodes, and the four PCs working together run 16 GASNet nodes. When splitting the Build and BFS tasks, we are spawning eight nodes per PC or 32 overall, so 16 nodes each work on the two tasks. To keep similar performance, we can distribute the two tasks in a cyclic manner. Since the two tasks run sequentially, half of the nodes will always be idle and barely impact performance.

Our goal for the heterogeneous version is to keep 16 Build nodes running in software and run 16 hardware BFS nodes on the FPGA. It is easier to build a routing topology in which the
software nodes and the FPGA nodes are contiguous halves of node IDs, so to emulate this, we are now distributing the tasks in a blocked manner. As a consequence, all the work for \textit{Build} is done on only two PCs with eight nodes each and all the \textit{BFS} work is done on the other two PCs, which leads to weaker performance for each task. However, we are only using this distribution as a means of migrating to the FPGAs. When starting the heterogeneous version, we are spawning 16 PC nodes again, all of which will run the \textit{Build} task, while the 16 FPGA nodes will run the subsequent \textit{BFS} task.

\subsection*{6.2.3 BFS hardware core}

The hardware implementation of breadth-first search has been written in VHDL. As any THe-GASNet hardware core, it needs to connect to an AXI bus as a burst master and to the GAScore through the four \textit{Fast Simplex Link} connections. We are again employing a PAMS to correctly sequence operation, communication and synchronization, leaving the raw data processing to the custom VHDL. Every PAMS in the system needs to be programmed with code sent through Active Messages. In the FPGA-linked version of the software, this code is being generated and sent by the control node 0.

Based on the same data structure used by the software version of BFS, the FPGA BFS core needs to execute the same two major tasks:

1. The adjacency lists of locally enqueued vertices need to be parsed, and edge information based on this needs to be sent to the remote nodes responsible for checking the other end of each edge.

2. Received edge information has to be processed by checking local vertices for parentage. Vertices without parent information are marked with the parent on the other end of the received edge and enqueued for the next search iteration.

These two tasks are logically independent and can be executed concurrently. This is being done in the software by switching occasionally between the two tasks (and would be a possible target for multi-threading); in the hardware implementation concurrency comes naturally. Figure 6.7 shows the structure of our BFS core. Task 1, in the top half of the picture, involves blocks for vertex dequeuing (\textit{ToDoListIn}), adjacency list parsing (\textit{VertexEdgeParser}) and accumulation of edges in buffers to send off (\textit{SendBuffer}). Task 2, in the bottom half, is composed of
processing the received buffers and providing a stream of edges (ReceiveBuffer), checking and amending local vertices’ parent information (ParentListCheck) and enqueuing newfound vertices (ToDoListOut). The only direct point of contact between the two concurrent parts is the exchange of local edge buffer information between the send and receive blocks. Edges involving local vertices on both ends are still accumulated in buffers, however the buffer is not copied, only the address is transferred to the receive block and given back after the buffer is emptied.

The only truly shared resource between blocks is the AXI bus to memory. There are three blocks reading from memory (ToDoListIn, VertexEdgeParser and ReceiveBuffer) and three blocks writing to memory (SendBuffer, ParentListCheck and ToDoListOut). Access to both the AXI read and write ports are assigned to the three users each through a least-recently used scheme. To promote contiguous DRAM bursts and limit row switching, bus access can be kept by a block for multiple bursts and is yielded again cooperatively.

Both tasks also communicate with the PAMS through control signals and FIFO-based data ports:

- PAMS initiates the start of a new BFS or a new iteration by control signal.
- SendBuffer writes buffer addresses to the PAMS, which it then uses to send off Long Active Messages.

\(^5\)ParentListCheck would in principle also have to read memory to check if a parent has been assigned, however, to limit random memory access we use a local on-chip scratchpad memory where previously found vertices are marked.
• PAMS writes addresses of received edge buffers to the *ReceiveBuffer* block.

• *ReceiveBuffer* writes addresses of emptied buffers to the PAMS, which it then redistributes with a *Short*-type Active Message.

• PAMS writes received addresses of emptied edge buffers back to *SendBuffer*.

• The end of an iteration is signaled to PAMS; it reads iteration statistics for traversed edges and found vertices from *ParentListCheck* and sends them with a *Short*-type Active Message to the control node.

Table 6.2 shows the resource usage for a single BFS core. As for the Jacobi core, BlockRAMs for buffering data dominate the resource requirements.

<table>
<thead>
<tr>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
</tr>
</thead>
<tbody>
<tr>
<td>4017 (0.58%)</td>
<td>5538 (1.61%)</td>
<td>106 (16.77%)</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

### 6.3 2D-DFT and stream encryption (ChaCha20)

As our third application case, we want to demonstrate both the integration of an existing IP core as well as the chaining of two different, independent operations. Our first chosen operation is a 2-dimensional *Discrete Fourier Transform* of images of up 16384*16384 pixels. Subsequently, we would like to encrypt the transform data with a modern stream cipher to be able to transmit it to a communication partner in secrecy. The basic flow is shown in Figure 6.8.

#### 6.3.1 Software-only implementation

**2D-FFT**

Practically, discrete Fourier transforms are most often implemented as *Fast Fourier Transforms* using the *Cooley-Turkey* algorithm. The most straightforward way to implement a two-dimensional discrete Fourier transform of an image with N*N pixels is to first apply the one-dimensional transform on each of the N rows of the image and subsequently apply the one-dimensional transform on each of the N columns of the result (obviously, doing the columns
first and and the rows second is equivalent)\(^6\).

When considering how to parallelize the initial run of one-dimensional FFTs on several cores or nodes, we observe that it is difficult to parallelize a single one-dimensional FFT, as the butterfly algorithm used in the Fast Fourier Transform would impose near-constant communication of intermediate results between parallel processors. However, splitting up the N image rows and assigning them to different processors to do complete 1D-FFTs on each of them is straightforward, and no communication between processors is required for this part. Uncompressed images will generally be stored in row major order in memory, meaning the pixel information in a row is a contiguous block in memory; vertically adjacent rows will also follow each other contiguously. Therefore, splitting up the work between several nodes also means splitting up the image vertically into contiguous blocks.

Applying the column part of the operation is suboptimal, however, as adjacent pixels along a column are not adjacent in memory, leading to much less efficient access of external DRAM. Furthermore, a single column is now split between multiple nodes, meaning the input data needs to be communicated from several places. Instead of making this communication part of the computation, we will instead transpose the result matrix of the first set of 1-dimensional transforms, as illustrated in Figure 6.9. By switching rows and columns, each result column of the first set of transforms becomes row-like contiguous memory data accessible by a single node. The same row-based one-dimensional FFT algorithms can now be used again to process this column data. Finally, to get back a correctly oriented 2D transform result, the data from the second set of FFTs has to be transposed again. The complete process is illustrated in Figure 6.10.

\(^6\)There is significant work examining alternative approaches to parallelize multi-dimensional Fourier transforms (e.g. [70][71]); we have chosen the documented approach as it allows a particularly straightforward demonstration of our programming model’s capabilities.
Instead of implementing our own one-dimensional FFT, we rely on the widely used *fftw* ("Fastest Fourier Transform in the West") library [72], which certainly outperforms any software code we could come up with. We are using a *single-precision* floating-point version of the library, as IEEE754 single-precision is the highest degree of precision available to us as an FPGA IP core. We are not really short-changing the CPU’s capability to do computations in double-precision floating-point, as using single-precision will double the pixel throughput of
both memory accesses and the CPU’s built-in vector instructions employed by *fftw*.

Implementing the transpose could be quite cumbersome, however it is already taken care of in our THeGASNet library by offering *Strided Accesses*: A transpose means nothing else but reading a contiguous block and writing single pixels with a stride of one line, or vice versa. We can specify both with our *StridedLong* Active Message type.

**ChaCha20 encryption**

*ChaCha20* is a symmetric stream cipher that has been proposed by Bernstein [73] as an improvement on his own Salsa20 stream cipher [74]. As part of RFC7539 [75], it has been proposed by *Google* for inclusion into the *Transport Layer Security* protocol for secure internet communication. It produces a *keystream* that provides an encryption byte for each unencrypted original byte in a file or data stream. The generation for each block of 64 encryption bytes is independent of previously generated data; each block generation differs by being seeded with a stream position. Therefore, encryption streams for different parts of a file or data stream can be generated in parallel. We are using ChaCha20 to encrypt the distributed 2D-FFT results in parallel.

Figure 6.11 illustrates how ChaCha20 generates encryption data. For every 512-bit block, a seed block of sixteen 32-bit words is assembled; its components are 128 bits of a fixed constant that is defined in the RFC document, a 256-bit symmetric encryption key, a 32-bit block count and a 96-bit nonce that is unique to a transmission session. A *round* of block generation involves segmenting the 16 words into four groups of four words and working on each of them in a so-called *quarter-round* operation. As shown on the left side of the figure, the quarter-round takes the four 32-bit values and re-combines them in a series of addition, antivalence and bit rotation operations. The grouping of words for the quarter-rounds is dependent on the number of the round and best illustrated by placing the 16 words in a 4*4 matrix as shown on the right side of the figure. In *even* rounds (counting from 0), the words are grouped in columns, in *odd* rounds they are grouped diagonally. ChaCha20 requires 20 rounds of generation; there are also weaker versions defined involving only 8 or 12 rounds that are still considered safe. As a final operation not shown in the figure, the 16 words generated after 20 rounds are combined word by word with the original 16 seed words through addition. The generated 64-byte encryption
Figure 6.11: Quarter-round operation on four 32-bit words (left side), block structure and encryption rounds on 4x4 words (right side)

stream can now be exclusive-ORed with the unencrypted data stream.

In our software-only solution, the encryption step can just be attached following the com-
pleted FFT, as it need only be applied to the local block of FFT spectral data.

6.3.2 Isolation of accelerator code and hardware modeling

For our hardware solution, we would like to implement separate FFT and encryption cores so we can try different combinations of software and hardware functions. To model this in software, we partition the FFT and encryption functions in separate GASNet nodes. In principle, this means an additional memory copy transporting FFT results in one node to an encryption node for further processing. However, since the second transpose involves a complete memory-to-memory copy anyway, we can as well make encryption nodes the destinations for the transpose’s strided Active Messages.

As the different software-hardware variations will include configurations that use hardware cores for both computations, we will separate out the control facility into a single software node that does not do computations.

6.3.3 Hardware cores

n*1D-FFT core

For the hardware FFT solution, we are also not designing our own implementation, but generating an appropriate IP core with the Xilinx Coregen tool. The hardware solution still follows the previous strategy of calculating one-dimensional FFTs on all of the rows that are part of the local data block. The core generator offers multiple options to balance area and resource requirements, precision and throughput for our 1D-FFT. As mentioned above, we chose single-precision floating-point as a data format for our FFT core, resulting in 8 bytes per pixel/data point for the real and imaginary part single-precision words. To maximize performance, we have further chosen the streaming variation of the FFT core, meaning that it can both accept a new data point and produce a new data point every cycle. To be able to fit up to four such FFT IP cores on a single FPGA, we have limited the number of data points to N=16384, resulting in a maximum (distributed) storage of N*N*8 Bytes = 2 GBytes per image.

We have packaged the generated IP core together with our standard PAMS core and memory burst components as shown in Figure 6.12. It will provide the core with the parameters for
FFT point size $N$ as well as the number of rows to compute for the core’s image block and start operation. Furthermore, it will control all Strided Active Message communications between FFT runs.

As we have a streaming core that will continuously accept new data, and as our image rows are a contiguous block of memory, our peripheral logic becomes quite simple: We need to stream the complete block of rows from its source address by AXI, and write a similar block of rows back to the destination address by AXI. The whole computation process becomes a streaming copy with added latency because of FFT processing. We have added 512-point FIFO data buffers at both the input and output sides of the FFT to smooth out any imbalances from competing burst accesses to the memory.

The completed 1-dimensional transform data will be transposed for the next processing step, however, this is not done by the core, but by GAScore itself through its Strided Active Message functionality.

**Encryption core**

The ChaCha20 encryption node shares the macroscopic structure of the FFT, as it also does a stream copy of a contiguous block of data. However, instead of processing an FFT of the data, it just executes an exclusive-OR operation off the data stream and a key stream that is generated directly in the core, as can be seen in Figure 6.13. Here we also use a PAMS to
provide parameters, specifically the size of the required streaming copy as well as the 256-bit encryption key, the 96-bit nonce and the 32-bit starting block count. With the encryption block parameters, a simple pipelined computation provides the 20 rounds of key mixing at a sufficient data rate to match the maximum streaming data rate.

Table 6.3 lists resource utilization for both the FFT and encryptions cores. The FFT is the only one of our application cores making significant use of the hardened DSP48 cores for its floating-point arithmetic. The ChaCha20 encryption mostly utilizes slice logic and registers.

Table 6.3: Post-map resource utilization for \textit{Nx1d-FFT} and \textit{ChaCha20} cores

<table>
<thead>
<tr>
<th></th>
<th>Registers</th>
<th>LUTs</th>
<th>BRAM36</th>
<th>DSP48</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Nx1d-FFT}</td>
<td>15197 (2.21%)</td>
<td>12493 (3.64%)</td>
<td>98.5 (15.59%)</td>
<td>57 (6.6%)</td>
</tr>
<tr>
<td>\textit{ChaCha20}</td>
<td>23500 (3.42%)</td>
<td>23561 (6.88%)</td>
<td>3 (0.47%)</td>
<td>-</td>
</tr>
</tbody>
</table>

(Percentages apply for a Xilinx Virtex-6-LX550T)

6.4 Application performance

In this section, we will report performance measurements on the software and hardware solutions we have implemented for our application use cases. We would like to strongly emphasize again that performance or performance gains are not the goal of this initial exploratory work. The following numbers should therefore be seen mostly as documented proof of application function,
as a calibration of where the current platform stands, and a guide towards understanding system behaviour for improved future systems to be built.

We do not aim to demonstrate acceleration gains from software to custom hardware, as this would not be seriously served by our arguably suboptimal implementations of both the software and the hardware solutions. Fine-tuning a specific application is not the goal of our current work, and we are aware of the limitations of our current system’s latency and bandwidth concerning memory access and networking; they will inform more ambitious future platforms we will build for further exploration.

Neither are we trying to prove that PGAS implementations show performance gains over other implementations. It is generally accepted that PGAS approaches make programming easier and more productive than other distributed models, but performance gains can only be achieved through extensive fine-tuning (which arguably voids some of the ease-of-use benefits). For a more extensive discussion of productivity and performance relating to our heterogeneous programming approach we refer to Section 7.5.

We are giving brief observations on the reported numbers including our assumptions about the performance differences. For a more thorough investigation geared towards fine-tuning performance and unambiguous identification of the particular bottlenecks, we would need to build more sophisticated profiling instrumentation building on the approaches used in Section 5.1.3. To repeat, this is not the focus of our current work.

### 6.4.1 Jacobi relaxation

Tables 6.4 to 6.6 show application throughput in Jacobi iterations per seconds for various configurations of PCs, FPGAs and nodes per device. The problem size has been increased proportional to the available memory per system. On each FPGA, configurations between two and eight nodes are viable, as previously illustrated in Figure 5.2. The minimum of two is suggested by the availability of two independent external memories and attached memory controllers. The maximum of eight is defined by the upper limit of each memory controller to logically split the memory to serve four AXI ports. This limitation is arbitrary and higher port numbers could be explored. We have mirrored the FPGA setups by also instantiating two to eight GASNet software nodes per PC, resulting in one half to two nodes per processor core.
What we can see from the limited number of results is that the Jacobi application is quite scalable; when increasing the problem size together with the number of PCs or FPGAs (i.e. weak scaling), throughput in iterations per second stays essentially the same. This is due to the fact that node-to-node communication takes less than 10% of the execution time even on the multi-device setups. Therefore, the local computation performance of each node dominates. We have not investigated in more depth while performance is essentially flat when splitting a single PC’s workload into two and four nodes, but increasing when using eight nodes. It is possible that splitting the data into smaller chunks per node benefits through cache capacity.

On the FPGAs, whose performance suffers from lower memory bandwidth than the PCs, we can observe that four nodes per FPGA (meaning two nodes per memory module) seem to start saturating the memory bandwidth. The PC results suggest that higher numbers of software nodes should be explored.

Table 6.4: Jacobi relaxation performance, 4096x4096 cells

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>iterations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PC, 2 computation nodes</td>
<td>10.63</td>
</tr>
<tr>
<td>1 PC, 4 computation nodes</td>
<td>10.46</td>
</tr>
<tr>
<td>1 PC, 8 computation nodes</td>
<td>13.93</td>
</tr>
<tr>
<td>1 FPGA, 2 computation nodes</td>
<td>4.99</td>
</tr>
<tr>
<td>1 FPGA, 4 computation nodes</td>
<td>9.13</td>
</tr>
<tr>
<td>1 FPGA, 8 computation nodes</td>
<td>9.33</td>
</tr>
</tbody>
</table>

Table 6.5: Jacobi relaxation performance, 8192x4096 cells

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>iterations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 PCs, 4 computation nodes</td>
<td>10.55</td>
</tr>
<tr>
<td>2 PCs, 8 computation nodes</td>
<td>10.43</td>
</tr>
<tr>
<td>2 PCs, 16 computation nodes</td>
<td>13.83</td>
</tr>
<tr>
<td>2 FPGAs, 4 computation nodes</td>
<td>5.03</td>
</tr>
<tr>
<td>2 FPGAs, 8 computation nodes</td>
<td>9.04</td>
</tr>
<tr>
<td>2 FPGAs, 16 computation nodes</td>
<td>9.33</td>
</tr>
</tbody>
</table>
Table 6.6: Jacobi relaxation performance, 8192x8192 cells

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>iterations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 PC, 8 computation nodes</td>
<td>10.49</td>
</tr>
<tr>
<td>4 PC, 16 computation nodes</td>
<td>10.21</td>
</tr>
<tr>
<td>4 PC, 32 computation nodes</td>
<td>13.03</td>
</tr>
<tr>
<td>4 FPGA, 8 computation nodes</td>
<td>4.93</td>
</tr>
<tr>
<td>4 FPGA, 16 computation nodes</td>
<td>9.03</td>
</tr>
<tr>
<td>4 FPGA, 32 computation nodes</td>
<td>9.13</td>
</tr>
</tbody>
</table>

6.4.2 Breadth First Search

For the Breadth-First-Search (BFS) application, we are measuring the number of (million) traversed edges per second, as suggested by the Graph500 benchmark. The problem size (measured in number of vertices: Scale is the power of two) is again increased when adding additional memory in the form of additional devices. A single PC’s memory can currently hold the graph data for about 4 million vertices (Scale 22). As is also defined by Graph500, the average number of edges that a vertex is connected to is 32.

We have again aimed for the two to eight nodes possible in a single FPGA. However, we are not able to fit an eight-node configuration into the area of our current FPGAs on account of the required BRAM resources. We have measured PC performance for configurations from two to eight nodes anyway. For a single PC, we get the best performance in a configuration of four software nodes, 68.5 MTEPS/s. However, as soon as we connect more than one PC, performance essentially flatlines around 30 MTEPS/s for any configuration. Our BFS implementation is very dependent on communication bandwidth because most newfound vertices need to be transmitted to other nodes for checking, and we suspect that the networking bandwidth of our Gigabit Ethernet fabric is becoming the bottleneck.

The FPGA components scale much better with multiple FPGA devices, most likely because the off-chip interconnect bandwidth between the FPGAs is much higher than for the Ethernet. However, overall performance starts off much weaker for FPGAs than for PCs. While we suspect low memory bandwidth as the culprit, the large difference requires further investigation.\textsuperscript{7}

\textsuperscript{7}We have explored a prototype memory controller optimized for contiguous burst operation and reached ca. 70 MTEPS on the four-FPGA system, however the prototype’s behaviour is still unstable.
Table 6.7: BFS performance at scale 22 (4.2 million vertices)

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>Mio. Traversed Edges/sec (MTEPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PC, 2 Build+BFS nodes</td>
<td>40.699</td>
</tr>
<tr>
<td>1 PC, 4 Build+BFS nodes</td>
<td>68.451</td>
</tr>
<tr>
<td>1 PC, 8 Build+BFS nodes</td>
<td>44.822</td>
</tr>
<tr>
<td>1 PC, 2 Build nodes - 1 FPGA, 2 BFS nodes</td>
<td>8.788</td>
</tr>
<tr>
<td>1 PC, 4 Build nodes - 1 FPGA, 4 BFS nodes</td>
<td>14.243</td>
</tr>
<tr>
<td>1 PC, 8 Build nodes - 1 FPGA, 8 BFS nodes</td>
<td>did not fit (too few BRAMs)</td>
</tr>
</tbody>
</table>

Table 6.8: BFS performance at scale 23 (8.4 million vertices)

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>Mio. Traversed Edges/sec (MTEPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 PCs, 4 Build+BFS nodes</td>
<td>28.628</td>
</tr>
<tr>
<td>2 PCs, 8 Build+BFS nodes</td>
<td>28.257</td>
</tr>
<tr>
<td>2 PCs, 16 Build+BFS nodes</td>
<td>26.402</td>
</tr>
<tr>
<td>2 PCs, 4 Build nodes - 2 FPGAs, 4 BFS nodes</td>
<td>16.375</td>
</tr>
<tr>
<td>2 PCs, 8 Build nodes - 2 FPGAs, 8 BFS nodes</td>
<td>25.331</td>
</tr>
<tr>
<td>2 PCs, 16 Build nodes - 2 FPGAs, 16 BFS nodes</td>
<td>did not fit (too few BRAMs)</td>
</tr>
</tbody>
</table>

Table 6.9: BFS performance at scale 24 (16.7 million vertices)

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>Mio. Traversed Edges/sec (MTEPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 PCs, 8 Build+BFS nodes</td>
<td>32.265</td>
</tr>
<tr>
<td>4 PCs, 16 Build+BFS nodes</td>
<td>30.904</td>
</tr>
<tr>
<td>4 PCs, 32 Build+BFS nodes</td>
<td>29.596</td>
</tr>
<tr>
<td>4 PCs, 8 Build nodes - 4 FPGAs, 8 BFS nodes</td>
<td>30.091</td>
</tr>
<tr>
<td>4 PCs, 16 Build nodes - 4 FPGAs, 16 BFS nodes</td>
<td>46.481</td>
</tr>
<tr>
<td>4 PCs, 32 Build nodes - 4 FPGAs, 32 BFS nodes</td>
<td>did not fit (too few BRAMs)</td>
</tr>
</tbody>
</table>

6.4.3 2D-FFT and encryption

Our last application demonstrates the chaining of two different functions, a 2D-FFT and an encryption with a symmetric ChaCha20 cipher. This application allows to arrange both software and hardware nodes in different combinations. We have done so for only one problem size and a limited number of node variations, as our main purpose here is to demonstrate the versatility of our system, not to assess or improve performance.

All measurements in Table 6.10 are for 4096x4096 point single-precision complex input data
(128MByte overall), which is set by the limits of our current Strided-type Active Message. The computations are always following the same sequence: 1D-FFTs of a horizontal strip of input data, a transpose of the result, another horizontal 1D-FFT that is effectively vertical because of the transpose, a reverse transpose and finally the encryption of the same-sized strip. We have measured the time for each of these steps separately. Overall, we have looked at five configurations, but for comparability each of them have eight nodes for FFT processing and eight nodes for encryption. Except for our baseline measurement, which is using our prototype code with eight nodes that each do both FFT as well as encryption, all measurements are done on four devices with four nodes each. Overall, there are four different combinations of all PCs, all FPGAs and mixed configurations.

Table 6.10: 4096x4096-point 2D-FFT + ChaCha20 encryption, all values are msecs

<table>
<thead>
<tr>
<th>Node configuration</th>
<th>FFT x</th>
<th>Transpose 1</th>
<th>FFT y</th>
<th>Transpose 2</th>
<th>Encryption</th>
<th>Sum 2D-FFT w/ Transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 PCs, 8xFFT+Encrypt</td>
<td>50.7</td>
<td>293.8</td>
<td>47.0</td>
<td>293.1</td>
<td>60.9</td>
<td>684.6</td>
</tr>
<tr>
<td>4 PCs, 8xFFT, 8xEncrypt</td>
<td>27.9</td>
<td>317.2</td>
<td>30.0</td>
<td>637.2</td>
<td>35.9</td>
<td>1012.4</td>
</tr>
<tr>
<td>2 PCs, 8xFFT 2 FPGAs, 8xEncrypt</td>
<td>27.8</td>
<td>291.3</td>
<td>29.4</td>
<td>862.5</td>
<td>124.4</td>
<td>1175.1</td>
</tr>
<tr>
<td>2 FPGAs, 8xFFT 2 PCs, 8xEncrypt</td>
<td>124.5</td>
<td>1019.1</td>
<td>124.6</td>
<td>191.4</td>
<td>36.4</td>
<td>1459.6</td>
</tr>
<tr>
<td>4 FPGAs, 8xFFT, 8xEncrypt</td>
<td>124.4</td>
<td>1021.6</td>
<td>124.5</td>
<td>818.9</td>
<td>124.4</td>
<td>2089.4</td>
</tr>
</tbody>
</table>

Several characteristics stand out to us:

- On any type of system, the required time for the transposes is much higher than the time used for the 1D-FFTs or the encryption. This puts into doubt the wisdom of our decision to compute the 2D-FFT via horizontal and vertical 1D-FFTs.

- When using only two PCs and running both the FFT and Encryption in the same nodes (first line of results), the actual FFT and encryption computations take longer than in separate threads on four PCs. This is unsurprising as they have double as much data
to process. However, the second transpose, which copies the result of the second set of FFTs into the memory location to be encrypted is faster on two PCs. This is because the four-PC solution results in all copies of the second transpose going across the network to a different PC (the node ordering means that FFT nodes are grouped together on the same PCs, and the encryption nodes together on others), while the first transpose is partially local in both setups.

- The software solutions again outperform the FPGA cores on both the 1D-FFT and the encryption. Both the FFT and the ChaCha20 encryption on the FPGA are fully pipelined and could sustain a throughput of nearly 400MB/s, resulting in a potential processing time of 40msecs for a 16MB data strip. As all the results involving them show nearly the same execution time of ca. 124.4 msecs, this indicates to us that memory bandwidth is again the limiting factor.

- Transposes involving writes to the FPGA are especially slow. Our transpose is executed by doing contiguous reads and strided writes, so the lower performance of our memory controller (especially in light of the required read-modify-write pattern) has a significant negative impact.

- As the second transpose is copied into the encryption nodes and not back into the originating FFT nodes, they tend to be slower, with only two FPGA-related exceptions.

Our overall takeaway from this experiment, however, is not the performance of certain components, but the ease with which it was possible to combine and re-combine the different software and hardware nodes.
Chapter 7

Why PGAS is a good fit for distributed FPGA systems

In light of our experiences building our framework and subsequently implementing applications on top of it, we will discuss in this section why we consider the PGAS programming model to be useful for programming and integrating FPGAs in distributed heterogeneous systems. As suggested in Section 1.2, this includes substituting the host-accelerator relationship with a non-hierarchical model of distributed heterogeneous nodes.

The successful implementation of the applications in Chapter 6 shows that it is possible to use PGAS in programming FPGA components; this section argues why it is a good idea. For that purpose, we will discuss the following general requirements for a distributed parallel programming model:

- **Data communication and communication cost exposure**: A parallel programming model for distributed systems must offer a mechanism to communicate data between nodes that are physically separate. It is preferable that this communication with remote nodes is made explicit so that the application programmer can take communication cost into account.

- **Synchronization**: A distributed programming model must offer mechanisms that allow nodes to synchronize, i.e. coordinate the initiation and completion of tasks to ensure operation on a consistent set of data.
• **Scalability:** A parallel programming model must offer tools to efficiently distribute an application both to a large as well as a variable number of nodes.

In addition to the points above, our **Unified Programming Model** approach of treating hardware nodes like software nodes implies a fourth requirement:

• **Migratability:** The programming model should allow the substitution of a software node with a hardware node without changes to the communication and synchronization patterns, and therefore without changes to any other node.

In each subsection, we will discuss two aspects of these requirements: How PGAS tackles them and why that approach works well for FPGA components in light of pivoting from the host-accelerator relationship to a model of heterogenous nodes.

### 7.1 Communication and cost exposure

The central feature of PGAS is the access to remote memories: In contrast to **distributed memory**/message-passing, reading or writing memory on another node can be executed without active involvement of the application thread or process running on that remote node. This property gives a PGAS node access to all memories in a distributed application that have been declared or opened up as shared memory, reflecting the **Global Address Space** in the programming model’s name.

However, similar to message-passing, accessing memory in another node requires an explicit remote write or read call. This property reflects the term **Partitioned:** A clear distinction between accessing local memory and remote memory is exposed to the programmer, so that the cost of a remote access in terms of network latency and bandwidth can be taken into account. This shared feature of PGAS and message-passing is obviously useful in modeling networked clusters of computers, but a similar case can be made for two other architectural variations.

Most FPGAs employed as reconfigurable computing hardware have their own memory. The majority of platforms take the form of PCIe cards, although autonomous FPGA hosts connected over Ethernet or other fabrics are entirely feasible, too (our own BEE4 platform could be connected this way). In both cases, high data access latency from and to an FPGA’s memory needs to be taken into account. As such, we conclude that the PGAS model is well-suited to
the data communication between software components on a common CPU and reconfigurable computing hardware components in an FPGA. Frequent communication of or access to small amounts of data is possible, but accurately modeled as expensive.

At the same time, utilizing the chosen PGAS API or language means that no accelerator-specific data transfers functions are necessary anymore. Like most implementations of the classic accelerator paradigm, efficient applications will try to limit data communication to large, bandwidth-exploiting bursts, often at the beginning and end of an “accelerated” computation. Unlike that approach, our implementation models these data transfers the same way as two software processes on different machines communicating. As a consequence, the communication pattern between software and reconfigurable computing hardware can first be modeled completely in software and then be adapted to hardware without changes to the API calls involved, as we will discuss further in Section 7.4.

7.2 Synchronization

Signaling conclusion of a data transfer, initiating a computation and communicating the completion of a workload are all tasks that are necessary both in distributed computing applications as well as when employing accelerators. Therefore, the extension of a distributed programming model to include “accelerators” as autonomous, heterogeneous nodes is a logical step for purposes of synchronization as much as it is for the data communication discussed above.

In fact, having the ability for data communication between nodes implies the potential for synchronization. While PGAS differs semantically from message-passing models because it defines communication as one-sided instead of two-sided, meaning its synchronization can be modeled as overlapping with computation on the remote nodes, it fulfills the synchronization requirement as well as other models.

Our specific communication implementation based on GASNet uses the concept of Active Messages; while this is not a required feature of PGAS, we have found it to be quite useful in our heterogeneous implementations. Because an Active Message implies execution of a handler function on the remote node, the required synchronization tasks can be actively triggered by this specific variation of a one-sided communication.
7.3 Scalability

To enable an application to scale across many parallel nodes, a programming model needs to provide the semantic means for a flexible division of data and workload. An illustrative example is the message-passing API MPI, which employs the following three features:

1. The acknowledgment - and assumption of - physically distributed memories that need to be explicitly connected through data transfers (messages).

2. The concept of Ranks. A rank represents a single computation entity, typically an operating system process. MPI's single-program, multiple-data (SPMD) programming paradigm means that with an individual rank number and awareness of the overall numbers of ranks, a process has all the information about which segment of the overall problem data it is supposed to work on and with which other ranks it has to communicate. This approach is principally independent of the actual number of ranks.

3. The concept of Communicators, which are subgroups of ranks. The option to partition all ranks in such subgroups opens the possibility of task parallelism in addition to the data parallelism implicit in SPMD. Communicators can be the scope of collective operations like barriers and reductions. A single rank can also be a member of several communicators, allowing for sophisticated work and data distribution schemes.

We have discussed the PGAS approach to the first aspect in Section 7.1.

The equivalent of Ranks exists under various names in various PGAS implementations. UPC, which also uses the SPMD execution model, employs the term Threads; GASNet designates single execution units as Nodes, and we are also using this designation through most of this text. Through our heterogeneous extension of GASNet, we have declared hardware as well as software components as nodes. We argue that this extends and enhances the originally software-centric scalability concept. Instead of software nodes with attached accelerators, we have software and hardware nodes across which various tasks and problems can be variably distributed. Furthermore, a cluster of software and hardware nodes need not be bound by the number of host PCs: Because of the agency our programming model gives FPGA components,
we can conceive heterogeneous clusters that are composed completely or for the most part of FPGAs.

Similar to the liberty to spawn a variable number of software threads or processes on a single CPU, a single FPGA can hold several GASNet nodes with equal or different capabilities. While the foremost limitation in this case is spatial, i.e. chip area, even this can conceivably be spread on a temporal scale through partial reconfiguration (both software and hardware share the limitations imposed by memory).

A significant precondition to the intended utilization of heterogeneous PGAS nodes is to also have an equivalent to MPI's *Communicators*, i.e. some kind of subgrouping capability to enable the assignment of different tasks to different types of nodes. We will discuss this requirement further in Chapter 8.

7.4 Task partitioning and migration

Our last requirement, migratability, is not covered by principal aspects of the PGAS model, but rather a manifestation of our concept of a unified programming model and common API for software and hardware components. As such, it is only a property of our specific heterogeneous implementation of the GASNet API.

We have illustrated in the applications section how a software-only implementation of an application can be prepared for FPGA migration by isolating the parts of the application slated for hardware implementation into separate GASNet nodes. This process of task partitioning has proved to be very beneficial to designing and integrating the hardware components at a later point.

Pure software algorithms reflect an attempt to write an efficient solution for CPU execution. When defining and writing the software nodes that are bound to be substituted by a FPGA component, the programmer is forced to reassess the structure of the code. The major benefit of the partitioning process comes in being able to identify the exact capabilities required of the FPGA core. The programmer needs to identify and minimize resource overlap between different parts of the code; data that needs to be available to several parts needs to be copied to each part if immutable, or communicated in the appropriate sequence if variable. Communication
and execution patterns need to be correctly sequenced, and required synchronization points need to be defined. External communication as well as internal execution can be assessed for FPGA viability. Potential obstacles for hardware implementation can be removed by reassigning certain functions to GASNet nodes that will stay implemented in software. Synchronization and communication can be modified while this still only involves moving a few barriers and function calls in C code.

Based on our application use cases, we have found the two-step process of partitioning and subsequent migration enormously helpful. With a working, partitioned software implementation at hand, subsequent steps to get a software-hardware solution operational usually did not involve trouble with communication and synchronization patterns, as they didn’t change with migration. Rather, the remaining bugs to be corrected in each application involved incorrect hardware design or an incorrect translation of the C communication patterns into PAMS code, both of which can best be avoided through a top-to-bottom high-level synthesis process, which we will discuss in Chapter 8.

7.5 Productivity gains vs. performance overhead for FPGA components

In the previous sections of this chapter, we have made a case why PGAS is an appropriate programming model for heterogeneous systems including FPGAs. Finally, we would like to look at two central questions when comparing our PGAS implementations on FPGAs with entirely custom-built solutions:

1. Does using PGAS/THeGASNet components in an FPGA design improve productivity?

2. Does using these components in an FPGA design impose a significant performance penalty?

We are discussing these two aspects together because they are two frequently competing qualities of a specific system and applications programmers need to weigh these trade-offs when deciding on a design platform. Despite the potential for performance gains through one-sided communication when using PGAS [76], the general consensus is that it can be competitive with the established message-passing models, but plays out its strength by providing significant gains
in programming productivity [77][78]. In a similar spirit, we argue that THeGASNet FPGA components significantly help productivity with insignificant performance losses. For this analysis, we will limit ourselves to the application parts completely implemented on one or more FPGAs, ignoring the benefits to any host-accelerator communication through our common API.

Any application involving transfers from one memory to another memory profits from the GAScore infrastructure because the programmer does not have to implement memory access and data communication from scratch anymore. What can still be considered trivial on a single system becomes very complex when data has to be transported to another FPGA, and even more complex if this is not constrained to a fixed number of FPGAs. The GASNet infrastructure provides scalability to communicate data in a system of any size.

A further benefit of our infrastructure involves the availability of *strided* and *vectored* remote data copies: Setting up the individual memory requests and fusing the data into packets is very cumbersome to implement but, in our case, entirely shouldered by GAScore.

Depending on the application, performance is either limited by *bandwidth/throughput* or access and communication *latency*, or a mix of both.

The impact of our infrastructure on *bandwidth* is limited: When doing bulk data transfers, each packet carries 2048 32-bit words of payload data. A *Long*-type Active Message adds a header of five 32-bit words (eight for strided messages) plus any arguments sent to the handler function. Even at the maximum number of 16 arguments, THeGASNet and NetIf communication overhead add only about 1% to the size of the packet and therefore impact bandwidth proportionally. This consideration assumes that network bandwidth is the bottleneck. Inside a single FPGA, remote copy bandwidth is currently limited by memory throughput, however this changes when several GASNet nodes share off-chip communication channels.

Obviously bandwidth suffers when nodes send many small amounts of data, like a single-word remote copy with a five-word header. However, in many cases this communication pattern can be avoided by packaging such data in *strided* and *vectored* accesses with the resulting accumulation of larger packets.

*Latency* is a more critical issue, as signaling or transmitting small amounts of data incurs significant delays through PAMS and GAScore processing and header latency (as this latency plays out in clock cycles, the exact impact on performance is, of course, dependent on the clock
speed of the components involved). As our measurements in Section 5.1.3 show, a single Short-
type Active message takes 20 cycles from PAMS to PAMS inside a single FPGA. It is easy to
conceive a custom solution that accomplishes the signaling between two on-chip components
in one or two cycles. Applications dependent on frequent back-and-forth signaling between
independent nodes would therefore not be well-served by our infrastructure. However, these
considerations shift again as soon as we extend systems past one FPGA and THeGASNet allows
signaling to any other node in a larger cluster.

When sending a single-word remote memory copy between two nodes in a single FPGA, our
measurements show an average latency of 76 cycles. Despite a few additional header words, at
this point memory access latency is already dominating the results again.

For, a more complete consideration of these effects, it is helpful to look at our application
examples:

- For the Jacobi/Stencil application, reasonably large local data sets result in much longer
  computation than communication times. Overall application performance is dominated
  by memory and compute bandwidth at a single node. Our communication requirements
  are well served by GAScore: Both the communication of contiguous memory portions to
  vertical neighbours as well as the strided memory communication to lateral neighbours
  are accomplished automatically by GAScore, saving us a lot of original design effort.

  For this communication pattern, bandwidth is dominating the performance. Overhead
  is therefore limited to the small packet headers, and strided memory data is efficiently
  packaged by a GAScore before transmission and later unpacked again by another GAScore.

- The Breadth-First-Search(BFS) involves a lot of communication of discovered edges to
  other nodes responsible for processing the connected vertices. All this edge information
  is packaged in the application core, so that only large packets are sent and memory
  and network bandwidth become the limiting factors. As a consequence, latency is of no
  significance to the application’s performance.

The BFS application illustrates a scalability benefit of using direct memory-to-memory
communication instead of message-passing. In terms of the algorithm’s semantics, sending
the edge lists as messages directly to the receiving application cores would work as
well; each receiving application core could directly process the incoming edges, and send off newly found edges as messages to others. However, delays in processing the messages would result in backpressure into the network and possibly even deadlock. This becomes especially complex to handle and resolve for larger clusters. To avoid this, intermediate buffering of received data in main memory is definitely preferable. While a message-passing system could surely implement this, PGAS semantics and THeGASNet infrastructure support this “out of the box”.

- The 2D-FFT and encryption application combines two distinct application cores, both of which are pipelined efficiently enough to process a memory word every cycle. As a result, this application is again memory-bound. Remote copy operations involve Strided-type Active Messages to transpose the 1D-FFT data as well as contiguous copies from the FFT results to the encryption nodes. Latency is again insignificant, and bandwidth overhead is minimal.

Because our implemented applications are all more dependent on bandwidth, we also want to discuss the hypothetical case of a latency-sensitive application:

- A Depth-First-Search (DFS) is the logical opposite to our Breadth-First-Search: Instead of finding all the neighbours of the current vertex, it jumps from vertex to vertex until it finds a vertex that has no further connections or that has been found before; only then, it will backtrack to the last vertex that had unexplored neighbours and continue to search as deep as possible from there. DFS is much harder to parallelize and essentially relies on a series of random memory accesses. If possible, it is best solved on a single machine, but large graph sizes and the resulting memory requirements can require that a cluster be used. In a cluster of considerable size, virtually every search step will lead to a different node; therefore, communication latency becomes critical.

In terms of our FPGA hardware, the significant latency incurred by our short messages becomes relevant for a Depth-First-Search, although each search step will also involve at least one memory read and the resulting memory access latency. Arguably an FPGA’s potential for high computation bandwidth is wasted on this application. However, if network
latency between FPGA nodes is smaller than between CPU hosts, it’s possible that an FPGA solution makes sense. In that case, minimizing our THeGASNet infrastructure’s latency could noticeably impact application performance.

### 7.5.1 Productivity and overhead in comparison with TMD-MPI

Since our work has been inspired by and is built on the same NetIf infrastructure as TMD-MPI [18], a *Message Passing Interface* implementation for FPGAs, we will compare in this section how application performance and design productivity differ by using THeGASNet.

The most similar use of TMD-MPI and THeGASNet is given when communicating data directly between two application cores. This is the standard communication model of TMD-MPI, while in THeGASNet this would constitute a *Medium*-type Active Message with *FIFO* bit set. GAScore will receive both the message request, in the form of a three-word header, as well as the payload data through the same FIFO. The *MPE* (Message Passing Engine), TMD-MPI’s equivalent to GAScore, has separate command and data channels and can therefore process the request header and data in parallel. Since TMD-MPI does not have a PAMS, a custom-written state machine must set up the *send* command, however all required data can be transmitted in a single command word. PAMS needs at least the number of header words in cycles to produce the request. GAScore then requires three additional cycles of latency before it will produce the first packet header word on the NetIf side. While we do not have current TMD-MPI performance numbers, we expect that MPE produces the first packet word four to five cycles faster.

On the receiving side, GAScore does not need to be made aware of incoming data, while the MPE has to get a *receive* command from the custom application state machine before data can be received. In both cases, the application core needs to have sufficient capacity to receive the data packet by FIFO.

We estimate that, overall, payload data could arrive up to ten cycles later through THeGASNet than through TMD-MPI. For latency-sensitive applications and small-message signaling, this could impact performance. For large-packet transmission and bandwidth-sensitive applications, differences will be negligible as the network bandwidth limits the performance and is therefore the same for both NetIf-using frameworks.
As we argued in the previous sections, a straightforward transmission of large amounts of data directly between two distributed application cores causes potential scalability problems, because backpressure could bring parts of a larger network to a temporary standstill. Our BFS application therefore transports lists of edges to be transmitted to another core directly from memory buffer to memory buffer, therefore making use of a much larger buffering capacity as is built into the NetIf fabric. For THeGASNet, this is a built-in functionality. When using TMD-MPI, the memory accesses on both sides need to be implemented as part of the custom application state machine. The exact complexity of this depends on multiple aspects:

- Are the memory accesses contiguous and of equal size or will data patterns vary?
- Is each memory accesses blocked or strided?
- Are the memory transfers statically schedulable or dynamically dependent on intermediate results?

All these factors influence the complexity and variability of both the memory bus accesses as well as the custom control flow to be implemented in the application core. Especially the third point has additional implications for TMD-MPI because the two-sided communication model means that every transfer has to be set up from both sides. It is safe to say that, for most applications, essentially re-implementing the functionality already provided by PAMS and GAScore will result in a minimum of several days of work and hundreds, possibly thousands of lines of HDL, including verification.

On the other hand, for such large-scale remote memory transfers patterns, it is unclear if much performance benefit can be preserved by TMD-MPI, as main memory latency will likely dominate over latency caused by PAMS and GAScore, and bandwidth will be limited by either the memory or the network, both of which would be the same for THeGASNet and TMD-MPI.
Chapter 8

Integrating FPGAs into PGAS

In the last chapter, we have illustrated that PGAS fulfills all the principal requirements for a scalable, productivity-oriented programming model that addresses heterogeneous distributed platforms. In this chapter, we discuss what additional requirements the specific properties of FPGAs and systems including FPGAs impose on a complete heterogeneous PGAS solution. We will explain what an ideal platform for PGAS with FPGAs should entail and outline our vision for a higher-level PGAS stack that is geared towards development of heterogeneous applications.

8.1 FPGA properties and resulting requirements

As customized, application-specific digital hardware, FPGA components have the capability to deliver high computation bandwidth and low energy-per-computation ratio. However, there are programming and system management aspects that are much harder to effectuate than in software.

8.1.1 Memory management

Most distributed applications and the communications runtimes they are built on can rely on an operating system that provides virtual memory management and dynamic memory allocation. If necessary, runtime libraries can add their own memory allocators on top of the operating system’s services. All these mechanisms can be elegantly and flexibly implemented in software without adding significant resource overhead. On the contrary, implementing similar
mechanisms in custom digital hardware can be complex and area-consuming. While significant research into this direction has been undertaken [79][80], the state of the art still assumes static memory allocation for most reconfigurable computing hardware.

When extending a specific PGAS API or language into heterogeneous systems, these limitations have to be taken into account. PGAS as a basic concept does not specify a certain memory addressing scheme. The original GASNet specifies an initial allocation of a shared memory segment, presumably through the host operating system’s mechanisms; the further allocation of memory resources in that segment is left to the language, library or application on top of GASNet. As another part of the specification, each GASNet node will hold a table with the virtual addresses and segment sizes of all other GASNet nodes in the distributed system.

We have diverged from this mechanism in our own THeGASNet implementation. Each FPGA GASNet node will have a pre-allocated memory segment of fixed size dependent on the resources available, sparing the need for dynamic allocation logic inside the FPGA. Furthermore, we have decided to do all remote memory communication based on address offsets relative to the start of the shared segment. Besides eliminating the need for a remote address table in every node (a scalability impediment, in our view), the offset addressing is essentially an address virtualization and implies restriction of remote memory accesses to the shared segment (a complete protection mechanism needs to also include checks against the size of the segment to restrict overflows).

Other communication runtimes or higher-level PGAS implementations might use other addressing mechanisms, but they will also need to acknowledge that, for the time being, FPGA components will likely keep relying on static, fixed-size allocations of available memory\(^1\).

### 8.1.2 Separation of data flow and high-level control flow

As we have emphasized before, the potential of FPGA components lies in the ability to create custom dataflows and pipelines tailored to the exact computation task required. This strength in modeling dataflows is contrasted by the difficulty to efficiently implement complex and flexible control flows that are quite simple to realize in software. Digital hardware can include finite

\(^1\)As future work will likely include virtualization mechanisms to replace, switch and move application nodes across existing hardware resources, we would predict that at least a basic flexible memory allocation mechanism for reconfigurable nodes will be required.
state machines, multiplexed and demultiplexed processing flows and efficiently unrolled and parallelized loops, but many data- and state-based decision mechanisms are much easier to shape in software. This includes the requirements of synchronization and remote data communication in distributed systems.

We have acknowledged and tackled this problem by adding a simple software-execution platform, the *Programmable Active Message Sequencer* (PAMS), to our designs. PAMS has become the interface between the dataflow-oriented computation logic and GAScore, our remote DMA engine that implements the GASNet Core API on the hardware side. PAMS does not have external memory access and does not take part in the actual computation; its only task is to synchronize with the internal computation and handle and initiate the external communication. While testing our hardware API infrastructure and implementing our applications, based on our application requirements it has grown from a simple programmable state machine to an application-specific processor with considerable flexibility. Pandit [50] has further advanced development by enabling interrupt-based handling of incoming Active Messages, an important extension to fully support the asynchronous, one-sided nature of our communication model.

Based on the positive experiences made with PAMS while implementing our application test cases, we strongly recommend a similar approach for any heterogeneous PGAS platform; area-intensive custom digital hardware should be limited to the performance-critical computation flow, a low-latency software solution can be used for the communication. It is important to emphasize that the software core should only be as complex as absolutely necessary for the required tasks. Any attempt to replace it with a full, universal, possibly multi-threaded processor is likely to unduly increase latency on computation and communication steps and will harm performance without adding real benefits.

For our sample applications, both the RTL and PAMS code for FPGA nodes were written by us from scratch. Such a manual migration process from software to hardware nodes has a built-in potential for errors as the outside behavior and sequence of communication might be accidentally changed. An ideal migration flow should therefore include an automated partitioning between tasks implemented in custom hardware and tasks to be executed by a PAMS-like software core. An example illustration of this process for a high-level PGAS language is shown in Figure 8.1. Automated compiler-like generation of PAMS code will not pose a great challenge; however,
8.1.3 Application launch

Even in homogeneous, software-only distributed systems, launching an application across a whole cluster is no trivial task. When implementing a runtime system for a heterogeneous platform including FPGAs, launching an application does not just involve spawning software processes and threads. It also requires the configuration of FPGAs with bitstreams. At the moment, our implementations still require pre-configuration of all bitstreams before launching the software components. On the MapleHoney cluster, bitstream configuration also has to be followed by a reboot of all PCs so that PCIe enumeration can take place.

A complete application launch infrastructure should include the ability to launch both software and hardware components based on three classes of provided data:

- One or more application binaries for the software nodes
- One or more bitstreams for the FPGA nodes
- Configuration data that describes the node assignments

Configuration data will not just need to include the assignments of different binaries and bitstreams to different node IDs and physical locations, but also requires information about the physical setup of host PCs and FPGAs in terms of the bitstream configuration flows. While we
have semantically decoupled hardware nodes from the host-accelerator relationship, the launch system still needs to know which FPGA to configure through which PC or which JTAG chain\(^2\).

### 8.1.4 Modeling heterogeneity at the application level

Our whole heterogeneous *Unified Programming Model* is centered on treating software and hardware components as nodes with equal agency to interact with other nodes. This is the foundation for our intent to migrate tasks from software to hardware easily. However, heterogeneity only makes sense if, at some point, we can fine-tune application performance by explicitly going through with that migration towards hardware components. This means that the application programmer needs awareness of the type of a specific node or group of nodes.

As we described in Section 7.3, the MPI library uses *Communicators* to build subgroups of processing nodes. Any PGAS language or library targeted towards heterogeneous platforms needs to include such a system of subgroups to be able to assign tasks to only one or the other type of node (task parallelism).

Furthermore, it needs to be able to *assemble* these groups, so there need to be node properties that let the application programmer check which nodes are FPGA nodes. Our current application implementations based on THeGASNet circumvent this necessity by assuming a predefined static distribution of software and FPGA nodes, but any truly scalable solution should be able to find these associations dynamically.

Among current PGAS solutions, *Unified Parallel C* has no concept of subgroups, so any task distribution runs counter to its basic SPMD programming paradigm, in which distributed arrays are spread out over all known nodes (called *threads* in UPC). On the other hand, *Global Arrays*, *Chapel* and *X10* all support subgrouping in some way and are therefore principally capable of supporting heterogeneous applications.

\(^2\)One of the developments conceivable for future systems is to make the configuration of application-specific bitstreams part of the THeGASNet infrastructure’s capabilities. As a precursor, we are currently programming application code into the PAMS through Active Messages. In an environment that utilizes *Partial Reconfiguration*, a statically pre-configured GAScore infrastructure could update application regions with bitstreams received through GASNet too.
8.2 PGAS requirements for forthcoming FPGA platforms

Recent commercial and academic efforts indicate a serious commitment to FPGAs as viable reconfigurable computing systems. New system architectures, infrastructures and connection topologies are being explored. In recently published research, for example, both Microsoft Catapult [81] and BlueDBM [82] use direct interconnect networks between FPGAs on multiple boards, despite also being connected to host PC architectures. These examples demonstrate FPGA accelerators as logically distinct units, with large private memories, that communicate with each other without constant involvement by host CPUs. To build large efficient clusters of these systems, a scalable programming model with FPGA components as independent actors is required. Our communications infrastructure fits these requirements quite well. In light of these developments, it is important to identify key requirements for future FPGA systems that support our programming model.

Both the Microsoft Catapult system as well as the OpenCL-based High-Level Synthesis solutions offered by Xilinx [9] and Altera [10] share the concept of a Shell: An outer layer of interface abstractions for off-chip memory as well as FPGA-to-FPGA, FPGA-to-PC and local area network connectivity. These abstractions layers are constantly kept in place, while specific applications can be switched in and out.

To optimize both throughput and area use of these Shells, it is under consideration to convert these interface abstractions into a hardwired layer instead of reconfigurable logic. For that, however, they should be generic enough to serve for various conceivable reconfigurable computing platforms. More experience is required before being able to determine what functions are best implemented in a shell.

From a PGAS perspective, the central support structure that we introduced is the Remote DMA unit, in our specific case GAScore, that joins off-chip connectivity, memory and the application-specific hardware. As we have discussed in Section 5.3, in our current systems we can implement our application and remote DMA if we have both an AXI interface for memory access and a NetIf interface for on- and off-chip connectivity.

---

3At this moment, only Xilinx offers Partial Reconfiguration of these application components in an openly available toolflow. Microsoft uses a proprietary partial reconfiguration flow; Altera is expected to introduce Partial Reconfiguration for their OpenCL flow on Stratix 10 devices in the near future.
Any FPGA platform that includes a “Shell” abstraction can be suitable for a PGAS application infrastructure if it offers both a memory bus and a connectivity interface. The important characteristic of our NetIf interface is that it has already abstracted away differences between a host-peripheral interface like PCIe, a custom FPGA-FPGA interface and a local area networking interface. Any abstraction that already uses the same data format for all three interface types inside the FPGA or allows easy conversion would also work for PGAS applications.

To decide on possible targets for hardening, our implementation results from Chapters 4 and 5 provide some guidance. Hard memory controllers (see Table 5.1) can provide considerable area savings and would be interesting for us if they provide an AXI interface. In terms of the on-chip network, a fully connected network for eight or more GASNet nodes would also save significant area by hardening (see Table 4.1). However, since state-of-the-art work would consider much larger numbers of possible endpoints [59], a fully connected hardened topology is unlikely to emerge.

Although our own off-chip communication has very low resource requirements (compare Table 5.3), local area network components with higher bandwidth and stronger error correction are likely to require sufficient resources to justify hardening. Our requirement for them is only that they plug right into the on-chip network to extend the same abstraction.

Our GAScore and PAMS components have reasonably small area requirements (see Tables 4.4 and 4.5), so we would recommend to keep them in the reconfigurable fabric. Aside from our examination of the best solution not having reached an endpoint, they are not guaranteed to emerge as a widely used, universal platform and would therefore, unlike memory controllers and networking, be considered a waste of resources for other design approaches.

### 8.3 THePaC++: A proposed library for exploration of high-level heterogeneous PGAS programming

While our current applications run directly on top the GASNet Core API, exploiting the advertised productivity advantages of PGAS implies using a higher-level language or application library. In light of the listed requirements for FPGA compatibility, we suggest that in the short term it would not be the best strategy to move forward by trying to adapt an existing high-level
PGAS language. We come to this conclusion for similar reasons as when deciding against using the existing GASNet implementation: It would place too many restrictions and compatibility requirements on what should be an early exploratory phase of completing the heterogeneous PGAS stack.

Instead, we suggest to write a new C++ library that PGAS applications can utilize. As a working title, we are calling it THePaC++ for Toronto Heterogeneous Parallel C++. Exploring our ideas in a C++ environment enables the use of a wide variety of development tools including more than one compiler infrastructure. We are also partial to the mix of performance-optimized C mechanisms and the code readability and software architecture benefits that well-used object-orientation can offer.

Before discussing the actual PGAS stack resulting from this approach, here are some of the features we envision for the C++ library. Few of these are original ideas; most are concepts that are implemented in at least one of Global Arrays, Chapel or X10. Naturally for C++, we would like to make substantial use of pairing of data and functionality in classes and objects, for example:

- **Nodes:** Making GASNet nodes encapsulated objects would allow a more natural grouping and data sharing between applications and handlers than the awkward shared-global variable approach we are using in our current Core API implementation. However, this is one specific aspect that would likely require a degree of vertical integration between THePaC++ and the THeGASNet layers below, in contrast to making THePaC++ a higher layer that is agnostic of THeGASNet internals.

- **Subgroups:** As previously mentioned, some notion of subgroups is fundamentally required to do useful task and data partitioning between heterogeneous platforms. In larger systems which are dependent on scalability and reliability in case of device failures, nodes and subgroups should likely carry a platform type property, so that work and data can be dynamically assigned to the appropriate available platforms. If regular communication between software and hardware nodes is required, information linking specific CPU-FPGA pairings, as in a PC hosting a PCIe FPGA card, might also be useful.

- **Distributed multi-dimensional arrays:** Data that is intended to be distributed on several
nodes should be automatically declarable as a single object. It needs to have an ownership property that assigns it to the group of nodes that hold the data. Various distributions like blocked, cyclic, block-cyclic need to be declarable in each dimension. Especially for vectors and matrices, operations like additions, multiplications and transposes should automatically lead to corresponding internal data transfers through TheGASNet. Similarly, if an array that is owned by one group is assigned to an array owned by another group, this should execute an automatic remote copy, as should a change of the ownership property. All these standard operations can be neatly overloaded on operators in C++.

- **Processing templates / design patterns:** Specific, commonly used data operations should also be encapsulated in an easily applicable way. An example for this could be the stencil operation used in our Jacobi application: A base class implements the basic optimized data access patterns needed in every kind of stencil operation, and the specific stencil computation for our problem can then be specified through inheritance.

The last two points in this list refer to distributed arrays and operations on them, typical aspects of a specific problem domain involving structured grids, which are important for physical simulations. The application example we implemented from this domain is the Jacobi relaxation.

Our approach to the C++ library is that it should be extensible for different problem domains. Our BFS application has very different requirements that should be accommodated with different classes and data structures. We believe that efficient implementations of these problems require that inside those encapsulated classes, we must be able to make direct use of features of the lower communication layers; however, users of the library, i.e. application programmers, should then be able to implement functionality without direct access to the lower layers. Modeling a computation pattern in a class can also lead to two different internal solutions, one optimized for CPU use and one for later HDL generation for FPGAs, as we will further discuss below.

Figure 8.2 shows the envisioned complete PAMS stack utilizing THePaC++. It has not changed in the network and GASNet Core API layers. On the software side, there is a generic Extended layer shown on top of the Core API. Whether this layer looks like the GASNet Extended API, has a different API and different features or is even possibly a part of THePaC++
for vertical integration purposes should be left open to exploration at this point. The software stack is topped by the C++ library and the application using the library.

On the FPGA side, our final goal is to generate all code above the GAScore from C++. That definitely includes generating both the computation core RTL as well as the PAMS code. We envision two possible approaches:

1. A conventional High-Level Synthesis approach combined with extraction of PAMS code: C/C++ code from both the application and the library source are converted into RTL for the computation core. The generated RTL needs to include the required memory access to AXI as well as the correct signals and data transfers to communicate with the PAMS. Furthermore, the HLS tool needs to be able to identify remote communication points in the code and generate PAMS code that interacts with the generated core. We previously illustrated this partitioning of code into sections to be implemented as PAMS code and sections to be implemented in hardware in Figure 8.1.

In reality, many operations, for example a matrix multiplication, could involve both a local computation and a remote communication component. It is also likely that certain execution decisions can only be made dynamically based on control flow, leading to additional software-hardware messaging. It is a non-trivial problem to guarantee efficiency in these cases.

High-Level Synthesis still has an uneven record for producing efficient code. Generally,
code needs to be annotated extensively with *pragmas* to lead to efficient pipeline structures and memory access patterns. This necessity to mold existing C code into the right patterns for efficient hardware generation informs our second approach.

2. The alternative approach of generating FPGA code involves a variation of *template metaprogramming* [83]: With fundamental hardware design knowledge, it is possible to model concurrency, memory access, dataflow and processing patterns that are likely to lead to efficient hardware structures. Designing a hardware core at the C++ level would now involve modeling these structures as C++ classes and chaining and combining corresponding objects in the same way as they would be connected in hardware. As these classes would reflect certain hardware design templates or even hardware modules, we have indicated them as a library layer called *HDL dataflow templates* on the FPGA side of our PAMS stack in Figure 8.2. A detailed application design would now lead to a similar 3-step process as the one we used in our C-based application use cases:

(a) Programming would start software-only based on the THePaC++ library. When the software solution is sufficiently matured, performance-critical code that has potential for FPGA speedup can be identified and isolated.

(b) The code segments targeted for FPGA hardware are modified or rewritten to model the required hardware dataflow with the hardware template class library. Each class in this library can be executed in two ways. First of all, it can be used as an exact functional, non-cycle-accurate, simulation of the future hardware components, therefore allowing verification of such a design in software-only.

(c) The second capability of each component would be to generate equivalent HDL code that can be synthesized into an RTL core. The hardware template class library would also include components modeling remote communication, so that PAMS code can be generated, too.

The illustrated process involves a lot more rewriting than out-of-the-box High-Level Synthesis, however it also likely to lead to more efficient results while keeping a well-structured programming style. This approach of modeling, functional simulation and late code gen-
eneration out of dataflows in a common computing language is, of course, not a new concept either: Both Streams-C [84] and FPGA Brook [85] are examples for similar concepts with promising results; more recently OpenSPL [86] based on Java syntax has been introduced. This design approach can also be hybridized by integrating High-Level Synthesis for smaller remaining code segments. We suspect that, in this case, the pre-structuring of dataflow with hardware-like components enables better performance outcomes than HLS alone.

Lastly, Figure 8.3 shows another perspective on the design process, visually separating the software stack and the hardware. At the bottom is the complete infrastructure, hardware computation nodes with PAMS as well as CPUs, all logically connected by THeGASNet infrastructure. Above, the software stack of THeGASNet, THePaC++ and the C++ application generates binary code for the software as well as RTL and PAMS code for the hardware. Option-
ally, for complex control flows the software code can dynamically generate and update PAMS code.

### 8.4 Summary of integration requirements and recommendations

In summary, based on our experiences building fully functional heterogeneous PGAS systems, we identify the following requirements to efficiently integrate FPGAs into a PGAS system:

- A heterogeneous runtime system should account for the limited memory management capabilities of FPGA hardware.

- It is advisable not to waste resources on implementing complex communication behaviour entirely in custom hardware. A limited software capability inside each GASNet node offers increased flexibility without a significant impact on performance.

- A heterogeneous runtime system needs the capabilities to launch software nodes and to configure FPGAs with the appropriate bitstreams.

- Any PGAS programming interface for a heterogeneous system needs to offer the programmer the capability to identify different types of node platforms and to group them together for different task assignments.

- Any partially pre-configured platform or platform with a hardened Shell with off-chip connectivity needs to offer memory and network abstractions that enable reconfigurable PGAS components to implement remote memory access.

- We recommend a high-level C++ library as a next exploratory step for heterogeneous PGAS applications and to evaluate the potential of a complete High-Level Synthesis flow.
Chapter 9

Conclusions

Still aided by Moore’s Law, the capacity and capabilities of FPGAs have steadily grown since their introduction three decades ago, and they have been established as viable accelerator platforms for large-scale computing and digital signal processing and as flexible, reconfigurable platforms for medium and large-scale embedded systems.

The growth in capabilities has not been accompanied by a corresponding improvement in design productivity. The vast amounts of digital logic resources are still widely utilized through cumbersome design processes involving low-level programming languages. Incorporating FPGAs into software design flows includes widely divergent, vendor-specific interfaces that are rarely part of the natural software programming model. Furthermore, there are few tools to express scalability across more than one FPGA device. Often reaching a second FPGA accelerator involves going through a PC host platform. This is especially unproductive when building large-scale clusters and distributed systems involving many FPGA accelerators.

We approach these problems with the underlying assumption that the mechanisms and models that allow distributed software components to communicate with each other are also useful to integrate accelerators into the software flow. Furthermore, we think that the fundamental interface to these accelerators should not be segmented by FPGA device. When a contemporary large-scale FPGA holds several homogeneous or heterogeneous computation units, each of these should be individually addressable in the programming model the same way an individual software component is.

Our work has been inspired by the ascent of a distributed programming model called Par-
Partitioned Global Address Space or PGAS, which is enabled by support infrastructure providing one-sided remote memory access, the ability to read and modify data belonging to another software component in a distributed system without involving that component’s program flow.

In our presented work, we have implemented a complete heterogeneous support infrastructure for one-sided communication, in the form of a library for software and in the form of hardware components for FPGA designs. Both forms of our infrastructure share a common API and allow CPUs as well as custom computation cores to utilize the same mechanism, making them interchangeable from the point of view of the infrastructure and the programming model.

We have demonstrated the usability of our infrastructure with three heterogeneous applications that started out as software-only. By using our common API, we were able to prepare and carry out substitution of software components through FPGA accelerators. The process of designing and migrating these accelerators has corroborated our initial concept and has given us valuable insight into the way forward towards a more complete platform.

Towards the further development of this idea, we have made the following contributions:

- We have provided the THeGASNet software library, a one-sided communication library that fully implements the same remote communication API for three CPU platforms, processors with x86-64 instruction set, Xilinx MicroBlaze processors and processors with the ARMv7 instruction set. THeGASNet is largely based on the existing GASNet Core API.

- To mirror the THeGASNet software capabilities, we have built GAScore (Global Address Space core), a hardware unit implementing remote memory access inside an FPGA. Similarly to a software node using the THeGASNet library, a hardware computation core in the FPGA utilizes a GAScore for all its communication with other software or hardware nodes.

- We have designed and provided PAMS (Programmable Active Message Sequencer), an area-efficient, application-specific processor that coordinates and sequences computation core function and GAScore operations. The underlying insight preceding its introduction is that utilization of all of GAScore capabilities is complex for a pure hardware design,
and that completely abandoning *software* inside the FPGA is not the most productive design approach.

- By building our infrastructure and implementing three applications, we have demonstrated that making FPGA accelerator components an integral part of a distributed shared memory programming model is feasible. Despite having used a fairly low-level communications API directly on both sides instead of implementing in a high-level PGAS language, we were able to illustrate a productive integration and migration process to FPGA components.

- We have collected valuable experiences in using a one-sided communication library for real applications, and have provided our analysis and recommendations towards building the next generation of heterogeneous, distributed systems supported by a complete, efficient, productive PGAS development flow.

Our work has been introduced to the FPGA and PGAS research communities through two full conference papers [87][88]. It has also enabled a recent M.A.Sc. thesis that leveraged and improved upon the infrastructure developed here and used it on a cluster of heterogeneous devices that combine an ARM system-on-chip with an FPGA [50].

Additionally, we published one more paper [89] on a design tool, *SimXMD*, which we wrote to be able to debug MicroBlaze software together with unverified hardware in simulation. *SimXMD* was also provided as *open source* to the FPGA community.

### 9.1 Future Work

We consider the presented work as only a starting point into a larger investigation of heterogeneous PGAS systems. In the following sections, we are introducing a few ideas how development could proceed.
Chapter 9. Conclusions

9.1.1 Short-term improvements

GAScore and PAMS

Despite fulfilling our requirements for the existing platforms and applications, both our central hardware components certainly have room for improvement.

GAScore implements the complete GASNet Core API Active Message functionality plus the added functions for strided and vectored transfers. However, it is not fully configurable yet in terms of memory and networking interface width. It could also be made more configurable for application-specific data access patterns and thereby improve latency in certain cases.

PAMS is strictly single-threaded and currently needs to be programmed with non-blocking polling mechanisms if multiple events can happen in unpredictable order, as in our Breadth-First-Search application. To fully support the asynchronous nature of the one-sided communication model, PAMS needs to be able to directly react to incoming messages of any type in any order, for example based on vectored interrupts. Sanket Pandit has implemented this functionality under the name xPAMS in his thesis research [50].

Finally, with both the PAMS and the GAScore architectures approaching maturity, it should be taken under consideration to integrate the two functions. Besides saving some added latency between the two components, integration would likely facilitate advanced functionality like atomic accesses. In terms of our development history, PAMS was a separate component because it substituted control flow functionality for GAScore communication that was easily implemented in software in MicroBlaze nodes. With ongoing development, migration of software to MicroBlaze nodes as an intermediate step has become less useful, therefore GAScore/PAMS could be turned into a fused block serving custom hardware cores only.

Networking protocols and fabrics

Currently, software communication is entirely TCP/IP-based for reasons of reliability and convenience. However, as TCP/IP is optimized more for bandwidth than latency, a move to a UDP-based protocol or even a custom layer on top of IP should be explored. Further exploration should naturally include RDMA-supporting fabrics like Infiniband [90] or RoCE [91] with their associated leaner protocol stacks.
To truly leverage the scalability that our programming model offers for hardware nodes, direct board-to-board FPGA connections need to be implemented for both platforms introduced here. The above-mentioned exploration of different communication protocols needs to consider how easy each of these can be implemented in hardware-only, something that is demonstrably complex for TCP/IP.

9.1.2 Long-term development

Heterogeneous Library: THePAC++

As introduced in Section 8.3, we consider a self-designed high-level PGAS library the best route to further explore heterogeneous applications written at a higher abstraction-level and to examine the potential of High-Level Synthesis.

GPGPU and MIC integration

As an important step towards a serious heterogeneous platform, a PGAS development flow needs to be extended to accommodate both General Purpose GPUs as well as the recently popular Many Integrated Core [52] architectures by Intel.

Profiling, performance prediction and autotuning

For our three use case applications, we have decided without supporting data which parts of the application - if not all of it - to migrate to hardware. A complete tool flow requires reliable tools to quantify performance bottlenecks in software. For a meaningful result, profiling cannot be limited to performance measurement on a single node but needs to include distributed communications in its assessment.

While this might well be a hard problem, we also recommend to explore the possibility of tools that make performance predictions before attempting a time-intensive hardware migration. At the other end of the implementation flow, finding the best mix of heterogeneous components through Autotuning [92] is a worthy research topic.
Partial reconfiguration and virtualization

We have already briefly touched on the subject of Partial Reconfiguration in Chapter 8. For a flexible platform, especially in a resource- or time-sharing context like an HPC cluster or a cloud, quick substitution of applications is desirable. Partial Reconfiguration is the closest FPGA approximation of the multi-tasking and multi-process capabilities of software operating systems. Depending on the system setup, an FPGA pre-configured with on-chip network and GAScores could limit reconfiguration to swap out applications as illustrated in Figure 9.1; alternatively a pre-configured “Shell” with only memory and off-chip-networking connectivity could host a variable number of GASNet nodes depending on area requirements.

A closely related subject is Virtualization, which is interesting for distributed systems for both reliability and scalability reasons. GASNet node IDs are an obvious venue for virtualizing addressing in a distributed cluster. GASNet nodes can be moved, replaced or duplicated with minimal infrastructure impact, and movement of the associated memory data is easily supported through the existing remote memory access mechanisms.
**Extending PGAS semantics with better local shared memory support**

As a goal for the overall PGAS community, but with benefits for FPGA as well as CPU systems, we suggest exploring high-level PGAS semantics supporting shared memory use among several PGAS “nodes” on a local platform.

Currently, many well-tuned distributed applications make use of hybrid programming models, for example a use of MPI or PGAS for the remote communication composed with *OpenMP* or *Pthreads* for multi-threaded shared memory computation locally. In our example software implementations, we have stuck with a single model and used the strict partitioned memory semantics of GASNet. To make best use of the available processing resources of a single multicore machine, we have run several GASNet nodes on a single PC. However, because we have stuck to the partitioned memory semantics, we have occasionally initiated memory copies where a shared access from several nodes would have been more efficient. Quite possibly, a better-performing solution would have been to run one GASNet node per PC and rely on Pthreads to use the multiple processor cores, similar to the popular MPI+OpenMP composition. However, this would have again increased the complexity of the programming model, and possibly made migration to the FPGA harder.

We have argued in this work for replacing another hybrid model - the combination of a distributed programming API and an accelerator API - with a universal distributed model that treats accelerator hardware as additional distributed nodes. Our hope is that a similar unification without loss of efficiency is possible for distributed vs. shared memory models. Higher-level PGAS languages like *Chapel* and *X10* already offer abstractions that allow the modeling of a computation as partially locally parallel and partially distributed, resulting in useful and efficient implementation options. Much of this can be accommodated through runtime support, for example by using *copy-on-write* mechanisms on data objects in the same way as an operating system does on process forks.

On a CPU system, the requirement for this hybrid approach exists because several computation units (e.g. processor cores) might best be programmed as separate, but they share one memory unit, which might be best modeled as shared. A similar requirement can exist on FPGA systems, for example when area resources allow the implementation of several processing
pipelines connected to a single memory resource. In light of our long-term vision of complete 
High-Level Synthesis flows, we hope that future high-level PGAS languages and libraries, es-
pecially for heterogeneous systems, accommodate this separate distribution of processing and 
memory resources.
Appendix A

THeGASNet implementation flow

A.1 Hardware flow

Figure A.1 illustrates the implementation flow for THeGASNet FPGA hardware from custom core development to download-ready bitstream. Provided components are shaded light grey, tools are shown in rounded, dark grey blocks and custom-created and generated files are shown white. Custom computation hardware (VHDL files in this example) can be combined with the PAMS code into a pcore (1), the IP core package format that Xilinx Platform Studio (XPS) uses. Outside connectivity for each custom computation core is limited to the Fast Simplex Links connections to the GAScore (through PAMS) and an AXI connection to main memory. Either custom hardware cores or MicroBlaze processors can be combined with THeGASNet infrastructure components like GAScore and NetIf as well as platform-specific hardware for memory and networking connectivity into a complete system, which is described in the proprietary *.mhs file format (2). Generation of MHS files is not automatized yet except for a command-line tool that builds the NetIf infrastructure. From this system description, Xilinx XPS can implement an FPGA bitstream (3), however this bitstream is still lacking NetIf routing information and firmware for MicroBlaze processors. A Python script named compile.py is responsible for the remaining steps. It is controlled by a configuration file that specifies all relevant information to build a complete THeGASNet system (4), specifically:

- The complete topology of a THeGASNet system, including the position and numbering of hardware and software (MicroBlaze) THeGASNet nodes on any FPGA (it is possible
Figure A.1: THeGASNet toolflow for hardware components. Provided components are shaded light grey, tools dark grey and custom-created and generated files are shown white.
Appendix A. THeGASNet implementation flow

164

to specify multi-FPGA and even multi-board systems).

- The specific application to run on each MicroBlaze node, including any compiler settings and application-specific command-line switches\(^1\).

- Routing table file paths for the system’s NetIf infrastructure.

- Any platform-specific settings, e.g. each target’s bitstream format.

Based on the configuration data, compile.py invokes the \textit{mb-gcc} cross-compiler to build all MicroBlaze applications and link them with the THeGASNet Core API library (5). The generated ELF binaries are then integrated into the FPGA bitstreams by invoking \textit{bitinit} (6). Similarly, routing table data for the \textit{Router_Init} pcore is added to the bitstreams through the \textit{data2mem} tool (7). Depending on the required configuration file format, the script can convert the generated bitstreams in the *.bit format into the *.bin format by invoking \textit{promgen} (8). At this point, the required files to configure all FPGA components in the system have been provided.

A.2 Software flow

Generating THeGASNet software applications for the \textit{x86-64} and \textit{ARMv7} architectures is much simpler than the hardware generation flow. THeGASNet provides a static library (\textit{the.gasnet-core.a}) to link against and a \textit{Makefile} to which new applications and required libraries can be added. The applications are then built directly on the intended Linux system by calling \textit{make}, which will then invoke the platform’s native \textit{gcc} compiler.

A.3 System launch

To launch heterogeneous THeGASNet systems, all the included FPGAs need to first be configured. We currently do not provide an automatic launch system that configures all FPGAs and starts all software nodes by invoking a single command. All FPGAs need to be individually configured and any THeGASNet software application needs to be launched once on each PC host (MicroBlaze software nodes are automatically started after bitstream configuration). If

\(^1\)As explained in Section 4.1, we have a mechanism to emulate command-line arguments for bare-metal MicroBlaze applications to keep the application code portable.
FPGAs are connected through PCIe, it might be necessary to reboot PCs first to trigger PCIe enumeration of the PCIe cores.

Launching a THegASNet software application on a PC or ARM system requires at least two command-line arguments:

1. 

2. 

An example command-line call to start a THegASNet application could look like this:

```
> ./thegasnet commtest --tgn myip=192.168.0.201 --tgn config=cluster4PCs4FPGAs
```

The configuration file has a very simple structure:

- Each line specifies one PC or ARM host in the system; all arguments in the line are separate by spaces. A hash symbol indicates a comment line.
- The first item on each line is the word HOST, followed by the host’s IP address (only IPv4 is supported at this point).
- The next item is the word THREADS, followed by two decimal numbers which indicate the range of GASNet node IDs that are launched on this host as software nodes. If the numbers are set to -1, there are no software nodes to be launched on this host.
- The node ID numbers are followed by two options: The first one is either set to BY_IP or to BY_FPGA and indicates whether messages to the listed software nodes should be routed through the FPGA NetIf infrastructure or through TCP/IP. It depends on the structure of the specific system if both options are feasible.
- The second option is either set to IP_LOOPBACK or to NO_LOOPBACK, where IP_LOOPBACK indicates that even communication between the local GASNet software nodes should be implemented through IP sockets (this is mostly a debugging feature).

The two options are followed by the keyword FPGA, followed by two decimal numbers indicated the range of hardware node IDs (or MicroBlaze software node IDs) found on
the FPGA connected to the host. It is again possible to indicate no FPGA nodes by setting the two numbers to \(-1\).

- The FPGA node numbers are again followed by the `BY_IP/BY_FPGA` option indicating whether messages to the nodes should by preference be routed by IP or through FPGA fabric.

An example configuration file could look like this:

```
## 4xPC+FPGA with four nodes each ##
HOST 192.168.0.201 THREADS 0 3 BY_IP NO_LOOPBACK FPGA 4 7 BY_FPGA
HOST 192.168.0.202 THREADS 8 11 BY_IP NO_LOOPBACK FPGA 12 15 BY_FPGA
HOST 192.168.0.203 THREADS 16 19 BY_IP NO_LOOPBACK FPGA 20 23 BY_FPGA
HOST 192.168.0.204 THREADS 24 27 BY_IP NO_LOOPBACK FPGA 28 31 BY_FPGA
```

This is an example for a node configuration feasible on our `MapleHoney` cluster: Four PCs with four software nodes each, connected with each other over a local area IP network, each one connected to an FPGA with four hardware nodes or MicroBlazes. The FPGAs are also directly connected through NetIfs, otherwise the `BY_FPGA` option would not make sense.
Appendix B

THeGASNet Data Formats

B.1 Active Message packet 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
- 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

<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></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>
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>Table B.2: Active Message parameters - <em>Strided Long</em> type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
</tr>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
</tr>
<tr>
<td>Contiguous block size in bytes (32)</td>
</tr>
<tr>
<td>Number of contiguous blocks (32)</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
</tr>
<tr>
<td>Stride in bytes between contiguous blocks (32)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table B.3: Active Message parameters - <em>Vectored Long</em> type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
</tr>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>Reserved (4)</td>
</tr>
<tr>
<td>Vector 1: Size in bytes (32)</td>
</tr>
<tr>
<td>Vector 1: Destination address, lower half (32)</td>
</tr>
<tr>
<td>Vector 1: Destination address, upper half (32)</td>
</tr>
<tr>
<td>(...) additional vectors</td>
</tr>
</tbody>
</table>

**B.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.
Table B.4: 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>Short, Medium, Long</td>
</tr>
<tr>
<td>#Arguments (8)</td>
<td>Short, Medium, Long</td>
</tr>
<tr>
<td>Destination/Token (16)</td>
<td>Short, Medium, Long</td>
</tr>
<tr>
<td>Handler ID (16)</td>
<td>Medium and Long only</td>
</tr>
<tr>
<td>Source (16)</td>
<td>Medium, Long (FIFO=0)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
<td>Medium, Long (FIFO=0)</td>
</tr>
<tr>
<td>Source address, lower half (32)</td>
<td>Long only</td>
</tr>
<tr>
<td>Source address, upper half (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>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 B.5: Active Message request parameters - *Strided Long* type

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Destination/Token (16)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Source (16)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
</tr>
<tr>
<td>Source address, lower half (32)</td>
</tr>
<tr>
<td>Source address, upper half (32)</td>
</tr>
<tr>
<td>Source: Stride in bytes between contiguous blocks (32)</td>
</tr>
<tr>
<td>Source: Contiguous block size in bytes (32)</td>
</tr>
<tr>
<td>Source: Number of contiguous blocks (32)</td>
</tr>
<tr>
<td>Destination address, lower half (32)</td>
</tr>
<tr>
<td>Destination address, upper half (32)</td>
</tr>
<tr>
<td>Destination: Stride in bytes between contiguous blocks (32)</td>
</tr>
<tr>
<td>Destination: Contiguous block size in bytes (32)</td>
</tr>
<tr>
<td>Destination: Number of contiguous blocks (32)</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
</tr>
</tbody>
</table>
### Table B.6: Active Message request parameters - *Vectored Long* type

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Destination/Token (16)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Source (16)</td>
</tr>
<tr>
<td>#Source vectors - 1 (4)</td>
</tr>
<tr>
<td>#Destination vectors - 1 (4)</td>
</tr>
<tr>
<td>Payload size in bytes (24)</td>
</tr>
<tr>
<td>Source vector 1: Size in bytes (32)</td>
</tr>
<tr>
<td>Source vector 1: Address, lower half (32)</td>
</tr>
<tr>
<td>Source vector 1: Address, upper half (32)</td>
</tr>
<tr>
<td>(⋯) additional source vectors</td>
</tr>
<tr>
<td>Destination vector 1: Size in bytes (32)</td>
</tr>
<tr>
<td>Destination vector 1: Address, lower half (32)</td>
</tr>
<tr>
<td>Destination vector 1: Address, upper half (32)</td>
</tr>
<tr>
<td>(⋯) additional destination vectors</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
</tr>
</tbody>
</table>
B.3 Active Message handler call 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 B.7: 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>Short, Medium, Long</td>
</tr>
<tr>
<td>#Arguments (8)</td>
<td></td>
</tr>
<tr>
<td>Handler ID (16)</td>
<td></td>
</tr>
<tr>
<td>Token (32)</td>
<td>Short, Medium, Long</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></td>
</tr>
<tr>
<td>0..512 words memory payload (32)</td>
<td>Medium only</td>
</tr>
</tbody>
</table>

Table B.8: Handler function call parameters - *Strided Long* type

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Token (32)</td>
</tr>
<tr>
<td>Payload size in bytes (32)</td>
</tr>
<tr>
<td>Contiguous block size in bytes (32)</td>
</tr>
<tr>
<td>Number of contiguous blocks (32)</td>
</tr>
<tr>
<td>Start address, lower half (32)</td>
</tr>
<tr>
<td>Start address, upper half (32)</td>
</tr>
<tr>
<td>Stride in bytes between contiguous blocks (32)</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
</tr>
</tbody>
</table>
Table B.9: Handler function call parameters - *Vectored Long* type

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM type (8)</td>
</tr>
<tr>
<td>#Arguments (8)</td>
</tr>
<tr>
<td>Handler ID (16)</td>
</tr>
<tr>
<td>Token (32)</td>
</tr>
<tr>
<td>Reserved (4)</td>
</tr>
<tr>
<td>#Vectors - 1 (4)</td>
</tr>
<tr>
<td>Payload size in bytes (24)</td>
</tr>
<tr>
<td>Vector 1: Size in bytes (32)</td>
</tr>
<tr>
<td>Vector 1: Address, lower half (32)</td>
</tr>
<tr>
<td>Vector 1: Address, upper half (32)</td>
</tr>
<tr>
<td>(...) additional vectors</td>
</tr>
<tr>
<td>0..16 Handler arguments (32)</td>
</tr>
</tbody>
</table>
Appendix C

Attributions

As some of our components are the work of others, we would like to clarify which code portions have not originated with us. We indicate where we are using a version that has been adapted by us in some form.

C.1 Manuel Saldaña and Arches Computing

The following components are originally the intellectual property of Manuel Saldaña and Arches Computing. They have been developed for Arches MPI and have been kindly provided in their original versions for our research:

- The NetIf and Router_Init peripheral cores for Xilinx EDK are providing the on-chip communication infrastructure for our FPGAs. NetIf has been moderately modified by us to allow more flexible routing; Router_Init is used unchanged.

- The FSL2IC peripheral core has originally been developed to connect the NetIf infrastructure with the onboard FPGA-to-FPGA connections of the BEE3. We have adapted the core to the BEE4, changed communication to DDR mode and added SECDED error correction.

- The compile.py Python script is used for MicroBlaze code compilation and bitstream generation and initialization. It has been extensively modified for the purposes of THegASNet and the BEE4 platform.
C.2 Sanket Pandit

The multithreaded THeGASNet Core API software library for Linux has been written entirely by us, but Sanket Pandit [50] has adapted it to reliably run atomic operations on the ARM platform and has added the Extended API implementation. All original hardware and modifications to GAScore and PAMS for the Zynq architecture and Zedboard have also been written by Sanket.

C.3 Others

- Xillybus Inc. have kindly provided their PCIe IP core netlist for free. We have encapsulated it in a Xilinx EDK peripheral core to connect it to the NetIf infrastructure.

- BEEcube has provided the BEE4_DDR3_MIG memory controller which connects to our self-designed AXI-compatible split memory controller.
Appendix D

Programmable Active Message Sequencer (PAMS) code

D.1 Opcode Table

A complete overview of PAMS instructions can be found on the following page. PAMS opcodes are 32 bits wide and composed of the following parts:

- A 5-bit opcode specifying the operation
- A control bit indicating that the following code memory word is still part of this instruction
- A control bit indicating that a variable number of following code memory words is part of this instruction
- A control bit indicating that an Active Message can be sent off after the current instruction
- A 4-bit code specifying a data source for the current operation (for ALU operations, data writes, and Active Message writes)
- 16 bits to specify various instruction-specific parameters
## Appendix D. Programmable Active Message Sequencer (PAMS) Code

### Hex Mnemonic Opcode +1 +x EndAM Source Description Parameters in opcode Extra words Comments

| 00 | PAMS_WAITFOR (PAMS_NOP) | 0b00000 | 0 | 0 | 0 | 15.0 | Wait for masked Ctrl inputs (4b) / MsgCtrs (4b) / TrnsfrCtr (4b) / Timer (2b) / AM return (1b) / Code loaded (1b) | Waits for all primed wait conditions in wait register to go 0. An all-zero word ("wait for nothing") implicitly means a NOP. |
| 0A | PAMS_TO_AMSEND | 0b00010 | 0 | 0 | 1/0 | x | | Write to AM send pipe from source (4b) / #Words (12b) | |
| 18 | PAMS_SETNODEID | 0b00011 | 0 | 0 | 0 | Node ID (16b) | Set Node ID |
| 24 | PAMS_ADJ_TIMER | 0b00100 | 1 | 0 | 0 | Offset (1w) | Correct Timer Offset |
| 2C | PAMS_TIMER_THR | 0b00101 | 1 | 0 | 0 | Value (1w) | Set Timer threshold Ctrl inputs (4b) / Threshold mask (2b) |
| 34 | PAMS_DATA_THR | 0b00110 | 1 | 0 | 0 | Value (1w) | Set Transfer Counter threshold Handler Code (8b) / Counter mask (4b) |
| 3C | PAMS_MSG_THR | 0b00111 | 1 | 0 | 0 | Value (1w) | Set Message-counter threshold Handler Code (8b) / Counter mask (4b) |
| 40 | PAMS_CTRLOUT | 0b01000 | 0 | 0 | 0 | Output mask (16b) | Pulse Ctrl Output  |
| 4A | PAMS_TO_CORE | 0b01001 | 0 | 0 | 1 | Output (1b) / #Words (12b) | Feed data to core (4b) / Output (1b) / #Words (12b) Words (x) |
| 52 | PAMS_ALU | 0b01010 | 0 | 0 | 0 | x | | Accumulate accumulated Operation (8b) / Accum (8b) / #Words (12b) | |
| 58 | PAMS_RESETWAIT | 0b01011 | 0 | 0 | 0 | | Reset waits flags Ctrl inputs (4b) / MsgCtrs (4b) / TrnsfrCtr (4b) / Timer (2b) / AM return (1b) / Code loaded (1b) |
| 60 | PAMS_SET_COUT_HCODE | 0b01100 | 0 | 0 | 0 | Handler Code (8b) / Ctrl output mask (4b) | Set's a handler code to trigger a certain output Handler Code (8b) / Ctrl output mask (4b) |
| 68 | PAMS_READPARAMS | 0b01101 | 0 | 0 | 0 | | Release rx_Hdr Data that isn't automatically dealt with by the other mechanisms Release rx_Hdr Data that isn't automatically dealt with by the other mechanisms |

### Accumulator operations | AM message handler codes

| 00 | Load | New code to put at address 0 + start(?) |
| 01 | Acc + Operand | New code to attach after current code (usefulness?) |
| 02 | Acc - Operand | New code to put at address |
| 03 | Acc AND Operand | New code to execute directly (interrupt regular execution) DirectReply: Short Messages whose arguments are fed right back to serve as remote reads |

| 240..247 | DirectReply: Short Messages whose arguments are fed right back to serve as remote reads |
| 254 | Don't save arrival time => follow up with sending last arrival time back |

### Data source codes

| 0 | Code Memory | 8 | Accumulator A |
| 1 | Timer | 9 | Accumulator B |
| 2 | Timer arrival register | 10 | Accumulator C |
| 3 | Core data port 0 | 11 | Accumulator D |
| 4 | Wait Vector | 12 | Core data port 1 |
| 5 | Node ID | 13 | Core data port 2 |
| 6 | Argument FIFO | 14 | Reserved |
| 7 | Memory Data FIFO | 15 | Handler ID |
D.2 PAMS code example

The following code excerpt demonstrates how our synthetic benchmark software generates PAMS code to measure the two-way Active Message latency in cycles between two FPGA nodes. It is framed as a C function executed on a PC THeGASNet node. The function receives arguments specifying the two THeGASNet node IDs and the FPGA cycle time at which to start the measurement. Using our predefined PAMS opcode macros, it fills an array with the appropriate instructions for the two involved nodes and then sends off each code set to the intended PAMS via a Medium-type Active Message. The data payload of Medium-type messages is not written into the FPGA memory, but instead transmitted to the custom hardware core. A specific handler ID (64..67) results in the data payload being automatically loaded into the PAMS’ private code memory.

```c
int32_t time_twowayshort(int src, int dst, uint32_t starttime)
{
    USE_NODE_GLOBALS(shared);

    uint32_t pamscode[PAMSCODESIZE];
    uint32_t pcaddr;
    int t;

    shared->received = 0;

    // CODE FOR INITIAL SENDER NODE ON FPGA (src) //

    for(t=0; t<PAMSCODESIZE; t++)
        pamscode[t] = 0;

    pcaddr = 0;
    pamscode[pcaddr++] = PAMS_WAITFOR(0, 0, 0, 0, 0, 1); // Wait for next code load
    pamscode[pcaddr++] = PAMS_NOP();
    pamscode[pcaddr++] = PAMS_SETNODEID(src);

    // Set up Message Counter 1 to wait for one returning message
    pamscode[pcaddr++] = PAMS_MSG_THR(hID_S0_ping, 1);
    pamscode[pcaddr++] = 1;

    // Set up time threshold to start measurement, wait for start time
    pamscode[pcaddr++] = PAMS_TIMER_THR(1);
    pamscode[pcaddr++] = (starttime);
    pamscode[pcaddr++] = PAMS_WAITFOR(0, 0, 0, 1, 0, 0);

    // Send Active Message; short asynchronous type, no arguments to receiving node
    pamscode[pcaddr++] = PAMS_TO_AMSEND(1, PAMSSRC_CODE, 2);
    pamscode[pcaddr++] = PAMSF_SHORT | PAMSF_ASYNC | dst;
    pamscode[pcaddr++] = ((hID_S0_ping<<16) | src;
```
Appendix D. Programmable Active Message Sequencer (PAMS) code

```c
// Wait for return message from other node
pamscode[pcaddr++] = PAMS_WAITFOR(0,1,0,0,0,0);

// Send Active Message; short asynchronous type, 1 argument
// delivers arrival time of returning message back to this PC node
pamscode[pcaddr++] = PAMS_TO_AMSEND(0,PAMSSRC_CODE,2);
pamscode[pcaddr++] = PAMSF_SHORT | PAMSF_ASYNC | (1<<16) | shared->mynode;
pamscode[pcaddr++] = ((hID_SI_deliver_timer)<<16) | src;
pamscode[pcaddr++] = PAMS_TO_AMSEND(1,PAMSSRC_TIMERARRIVAL,1);

// Jump to 0, make PAMS wait for next code load
pamscode[pcaddr++] = PAMS_BRANCH(PAMSFLAG_ALWAYS,0);

// Program initial sender node with the PAMS code generated above
gasnet_AMRequestMedium0(src, hID_M0_newcode_0, (void*)pamscode, pcaddr<<2);

// CODE FOR INITIAL RECEIVER NODE ON FPGA (dst) //
for(t=0;t<PAMSCODESIZE;t++)
  pamscode[t] = 0;
pcaddr = 0;
pamscode[pcaddr++] = PAMS_WAITFOR(0,0,0,0,0,1); // Wait for next code load
pamscode[pcaddr++] = PAMS_NOP();
pamscode[pcaddr++] = PAMS_SETNODEID(src);

// Set up Message Counter 1 for one initial message; wait for message
pamscode[pcaddr++] = PAMS_MSG_THR(hID_S0_ping,1);
pamscode[pcaddr++] = 1;
pamscode[pcaddr++] = PAMS_WAITFOR(0,1,0,0,0,0);

// Send Active Message; short asynchronous type, back to original node
pamscode[pcaddr++] = PAMS_TO_AMSEND(1,PAMSSRC_CODE,2);
pamscode[pcaddr++] = PAMSF_SHORT | PAMSF_ASYNC | src;
pamscode[pcaddr++] = ((hID_SI_ping_timer)<<16) | dst;

// Jump to 0, make PAMS wait for next code load
pamscode[pcaddr++] = PAMS_BRANCH(PAMSFLAG_ALWAYS,0);

// Program initial receiver node with the PAMS code generated above
gasnet_AMRequestMedium0(dst, hID_M0_newcode_0, (void*)pamscode, pcaddr<<2);

// WAIT FOR TIMER DATA TO ARRIVE FROM FPGA NODE
while(!(READ_SHARED(shared->received)));
shared->received = 0;

// CALCULATE AND RETURN TIME DIFFERENCE
return READ_SHARED(shared->timer0) - starttime;
```
Bibliography


specifications

graphs: An approach to modeling networks,” Journal of Machine Learning Research,


[70] H. M. Buecker, “Row-column versus vector-radix: A case study of two parallel 2D Fourier
transform algorithms,” Parallel and Distributed Scientific and Engineering Computing:

Fourier Transform using 2D domain decomposition,” Parallel Computing, vol. 39, no. 1,

Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal

[73] D. J. Bernstein, “ChaCha, a variant of Salsa20,” in Workshop Record of SASC, vol. 8,
2008.

[74] ——, “The Salsa20 family of stream ciphers,” in New stream cipher designs. Springer,


[76] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, “Optimizing bandwidth limited prob-
lems using one-sided communication and overlap,” in Proceedings 20th IEEE International
Parallel Distributed Processing Symposium, April 2006, pp. 10 pp.–.


