A Remote Memory Access Infrastructure for Global Address Space Programming Models in FPGAs

Ruediger Willenberg
Dept. of Electrical & Computer Engineering
University of Toronto
Toronto, Ontario, Canada
willenbe@eecg.toronto.edu

Paul Chow
Dept. of Electrical & Computer Engineering
University of Toronto
Toronto, Ontario, Canada
pc@eecg.toronto.edu

ABSTRACT

We are proposing a shared-memory communication infrastructure that provides a common parallel programming interface for FPGA and CPU components in a heterogeneous system. Our intent is to ease the integration of reconfigurable hardware into parallel programming models like Partitioned Global Address Space (PGAS). For this purpose, we introduce a remote memory access component based on Active Messages that implements the core API of the Berkeley GASNet communication library, and a simple controller that manages communication and synchronization for custom FPGA cores. We demonstrate how these components deliver a simple and easily configurable communication mechanism between distributed memories in a multi-FPGA system with processors as well as custom hardware nodes.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/Software interfaces*; C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Adaptable Architectures*; D.1.3 [**Software**]: Programming Techniques—*Parallel Programming*

General Terms

Design Performance Languages

Keywords

Parallel Programming Models; FPGA; PGAS; RDMA

1. MOTIVATION

High-Performance Reconfigurable Computing (HPRC) systems present two main challenges to application programmers: What parallel programming model to use, and how to incorporate reconfigurable hardware into a software application.

FPGA'13, February 11–13, 2013, Monterey, California, USA.

The first problem, inherent to all distributed computing, is what model of the existing hardware and memory distribution to present to the application programmer. This has implications for how to distribute and communicate data across the system, how to synchronize computations, and for how explicitly the programmer has to consider the physical makeup of the system. On the one end, *Shared Memory* presents a unified address space to the programmer, similar to the one found on a single host. On the other end, *Distributed Memory* only lets the programmer access the local memory, and all data exchange with other nodes happens explicitly through communication called *Message Passing*. The shared memory model is easy to program, but often leads to inefficient code, since the compiler cannot sufficiently reason about data access and communication patterns. The distributed model can produce very efficient implementations, but is very cumbersome to program.

The second problem involves the fact that most high-performance application programmers understand software and CPU-based systems, but not reconfigurable hardware. Part of that problem is being attacked by emerging tools to translate high-level language CPU code into Register-Transfer Language, with mixed results so far. However, besides an automatic synthesis path, applications also require an infrastructure for communication between software and hardware computation nodes, the equivalent of a communication API between CPU hosts. Preferably, this infrastructure should be independent from specific FPGA platforms, given the multitude of concepts and products that connect FPGAs with CPU-based host systems.

Both problems presented above point to the larger issue of increasing software and hardware complexity. Performance and efficiency are still the most common metrics for computing systems, but *productivity*, as measured by the required effort to design, debug and maintain high-performance computing applications, has been recognized as essential to continued progress towards exascale systems [17].

In our opinion, a unified programming model and API for all components in a heterogeneous system (see Figure 1) is crucial to keeping applications maintainable and scalable. Furthermore, the prototyping of algorithms in software and the subsequent migration to hardware accelerators is facilitated by such a common API.

In this paper, we will present our vision of a C++-based application design process that is based on the *Partitioned Global Address Space* model (PGAS). As our main contribution, we introduce an FPGA communication infrastructure compatible to GASNet[12], an existing PGAS communica-

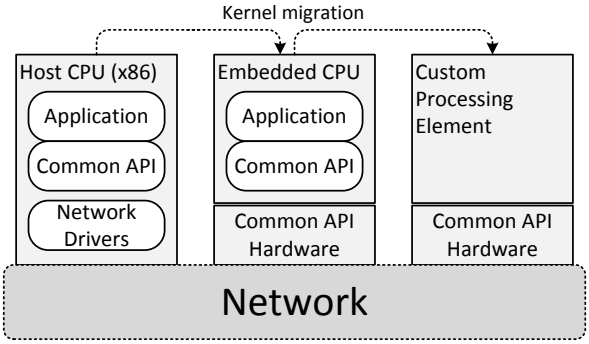

Figure 1: Example for a multiple-platform system with unified parallel API: Host CPU, embedded FPGA processor and custom FPGA component

tion API that maps well to FPGA as well as software components. This way, we enable cross-platform communication as well as easier software-hardware migration. The two main building blocks of this infrastructure are:

1. GAScore (Global Address Space core), a component that manages the remote memory communication for embedded processors and dedicated hardware processing elements. GAScore implements a GASNet-based API for embedded processors and custom hardware cores.
2. PAMS (Programmable Active Message Sequencer), a small communication controller that manages messaging, synchronization and GAScore access for custom hardware cores.

This paper is organized as follows: In Section 2, we give a bit of background on parallel programming models and the GASNet library on which our work is based. Section 3 introduces the hardware cores we developed in detail, and Section 4 explains the software stack that we plan to build on top of these components. Section 5 examines current system performance. Related work is referenced in Section 6. Section 7 elaborates on our next steps, and Section 8 concludes.

2. BACKGROUND

While our presented work is centered on implementation of a hardware communication core, it is important to understand the larger agenda that justifies this effort. Therefore, we will briefly introduce the parallel programming model that our hardware accommodates. We will also give a quick overview of the GASNet communication API that our design closely models.

2.1 Partitioned Global Address Space model

The two basic memory models for distributed parallel programming are called *shared memory* and *distributed memory* (better known by its practical implementation, *message-passing*). Shared memory assumes on the application level that all memory locations in the distributed system are directly accessible in a single address space. Accesses to physically remote memories need to be implemented by a runtime library. This library also needs to manage memory consistency and coherency. While the opacity of physical memory distribution eases programming of applications with this model, it can lead to inefficient data distribution and synchronization overhead. The most popular implementation of shared-memory parallel programming is the language and compiler extension *OpenMP*[5].

In distributed memory models, no direct access to remote memory is possible. Data needs to be communicated through the passing of messages, which will then be correctly related to local memory by the local process. Communication is implicitly two-sided, with both sender and receiver having to call functions for a transfer to be initiated. Message-passing can be fine-tuned for efficiency, but is more complex to implement. Furthermore, with the increasing size of computing clusters, the two-sided communication approach becomes a bottleneck for scalability. The dominant standard for message-passing is the *Message Passing Interface*[4] (MPI).

Partitioned Global Address Space (PGAS) offers a tradeoff between the two models described above. Every location in shared memory is directly accessible by any processing node. However, PGAS languages and libraries offer an explicit distinction between local and shared memory. Remote memory accesses are one-sided: When the memory of a remote node is accessed, no implicit synchronisation with the computation process on that node is happening. This is different from message-passing, where communication is always two-sided[1]. As a consequence of these attributes, PGAS offers the following advantages:

- Programmers have better awareness of the cost of a memory operation.
- The explicit designation of remote memory enables a relaxed use of consistency and coherency, and therefore avoids redundant synchronization overhead.
- One-sided access allows better scalability on large systems.

Languages that implement this programming model include the dialects Unified Parallel C (UPC)[8], Co-Array Fortran (CAF)[1] and the Java-based Titanium[7], as well as the newly designed parallel idioms Chapel[13] and X10[14]. SHMEM[6] and Global Arrays[20] are application libraries making use of the PGAS model.

2.2 GASNet communication library

GASNet (Global Address Space Networking) has been specified as a remote communication library for the Berkeley UPC and Titanium languages. Its API structure acknowledges, but does not require, the capabilities of *Remote Direct Memory Access* (RDMA) networking hardware like *Infiniband*[3]. GASNet's *Core API* is based on the principle of *Active Messages*[23]. Active messages are essentially asynchronous remote procedure calls. They are initiated by calling a *Request* function. The call defines the source address to copy from, the number of bytes to copy and the target address on the remote node. Besides payload data, an active message always includes a handler code and handler arguments. The handler code specifies which function to call on the remote target when the message arrives, and the arguments are handed to that function. Handler functions can fulfill synchronization purposes, process the arrived data in some way or initiate replies. An active message request does

[1]MPI 2.0 has heavily constrained one-sided Remote Memory Access support, but this is not part of the message-passing paradigm. For a detailed critique of its limitations, see [11].

not require an answer. However, if an answer is required, for example in a remote memory read, a handler function is allowed to initiate exactly one *Reply* message, which can only go back to the requesting node. These constraints safeguard against deadlocks.

Core API Active Messages are limited to packet sizes that can be easily supported by network hardware. The GASNet *Extended API* offers transfer functions for unlimited sizes and several types of barrier synchronization. It can be implemented entirely through Core API calls. However, sophisticated RDMA networking hardware can directly support Extended API functions.

3. SYSTEM OVERVIEW

In general, parallel programming models and APIs can be accommodated to a variety of infrastructures, so consequently MPI can be run on top of GASNet as well as GASNet can be run on top of MPI. Both of them are being able to communicate over regular TCP/IP-based networking stacks or Remote DMA hardware like Infiniband. However, each of these translations through software stacks costs performance. We have therefore concluded that dedicated hardware support for our chosen API GASNet is essential to maintain low latency, a crucial metric for parallel systems. Consequently, our main communication component uses a control API very close to GASNet and directly implements its Active Message capability for memory-to-memory transfers.

3.1 GAScore structure and use

The *GAScore* (Global Address Space core) is an implementation of GASNet functionality in hardware form. Processing nodes in FPGA systems that use GAScore are composed as shown in Figure 2. A computing element in the form of an embedded processor or a hardware processing engine is connected to one port of a dual-ported local memory (BlockRAM). The second memory port is connected to the GAScore. The GAScore is connected to the on-chip network.

The computing element is connected to the GAScore with four Fast Simplex Links (FSLs), essentially Xilinx-specific 32-bit-wide FIFOs. Figure 3 takes a closer look at the internal structure of the GAScore and illustrates the purpose of the four connections in receiving and transmitting Active Messages.

If an Active Message packet arrives from the on-chip network (see lower right corner of Figure 3), it is processed by the *Receive* unit. If the message holds a data payload, that payload is first written to the intended memory location specified in the message parameters. Because of the previously described deadlock-avoidance constraints, the receiving processing node never learns the sender's node address. Instead, the source node address is written into a *Token buffer*, which returns a token code as a key to the stored node address. The token and further parameters and arguments are transmitted over FSL 1 to the computing element, thereby calling the intended *handler function*. Whenever the handler function completes, it returns the token through FSL 2 and the Token buffer can free the stored source node address.

The computing element can request the sending of Active Messages by writing any message parameters and handler function arguments over FSL 3. Unless the request is for a short message without payload, the *Transmit* unit reads

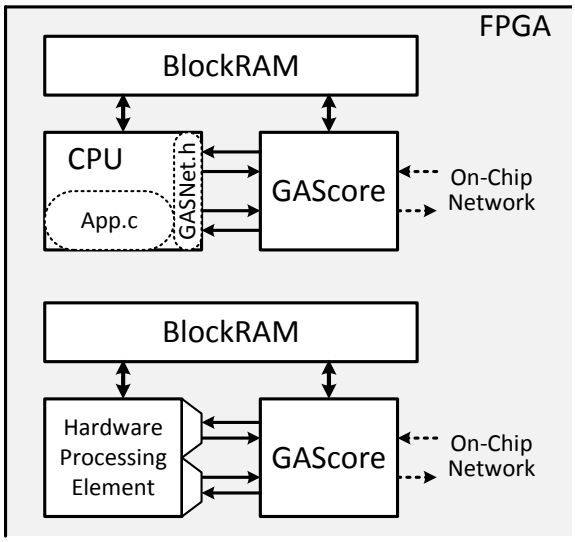

Figure 2: On-chip GAScore configurations

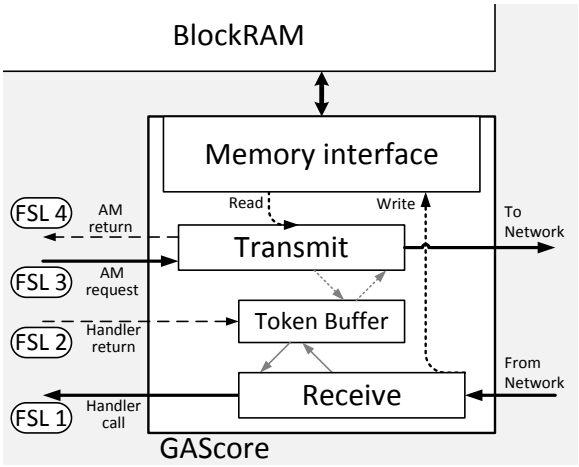

Figure 3: Internal GAScore structure

data from memory. This data, message parameters and handler function arguments are combined into an Active Message to be sent out over the network. A message can either be a new request, or it can be a handler function's reply to a prior request, as in the case of a remote read. In the first case, the computing node includes a destination node with the parameters. In the second case, the handler function includes the priorly received token. The token is used to look up the destination address in the Token Buffer. When the message has been sent, the GAScore signals the completion of the message request over FSL 4. This informs the computing element that the data to be transmitted has been read, and that the related memory area can now be freely operated on again.

While the data flow between computing node and GAScore could be multiplexed into two links, transmission and reception have explicitly been kept independent so that no deadlocks can occur. The only shared components are the Token Buffer, which needs no locking, and the memory bus, which is shared round-robin between reads and writes.

Table 1: GAScore synthesis statistics, Xilinx XC5VLX155T-1

Resource	#	%
Registers	760	0.78
LUTs	1336	1.37
BRAMs	2	0.47

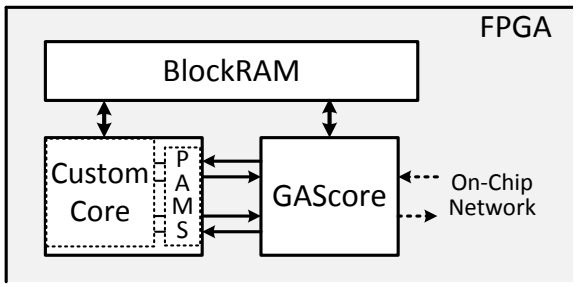

Figure 4: GAScore configuration with custom core and PAMS

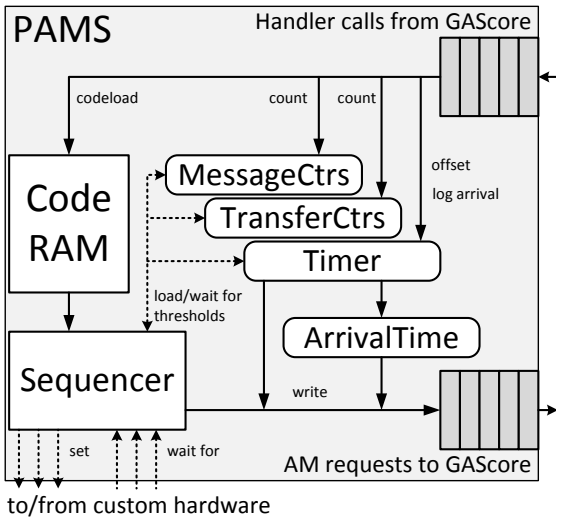

Figure 5: PAMS overview

Table 1 shows the resource utilization at a maximum clock speed of 142.6MHz, projected by synthesis for a GAScore with BRAM interface on a Xilinx XC5VLX155T-1. While synthesis numbers are not as reliable as final implementation numbers, we can see that a single GAScore uses a very small amount of chip resources.

3.2 Programmable Active Message Sequencer

For the Xilinx Microblaze embedded processor, a GASNet Core API library implementation has been developed that translates to and from the GAScore FSL channels (see Figure 2). Because the parameter format and GAScore functionality have been so closely modeled on the GASNet specification, translation is very simple and efficient. We are convinced that the code can be easily adapted to other embedded processor architectures.

On the other hand, custom hardware cores can implement their own mechanisms to send Active Message requests to the GAScore and process Handler function calls. However, this would be fairly redundant work for different computation cores, and individual implementations would make incompatibilites more likely.

Instead, we have developed a small application-specific controller that executes a limited set of machine code instructions, called the *Programmable Active Message Sequencer* (PAMS), to control the GAScore communication and synchronize it with core computations. Figure 4 shows a custom core/GAScore combination with the PAMS. A structural overview of the PAMS can be seen in Figure 5. The sequencer has the following features:

- PAMS code can be loaded and re-loaded into the sequencer instruction RAM through specialized Active Messages.
- Message counters (MessageCtrs) can be configured to count messages of a specific handler code and indicate when a threshold is reached. This is useful, for example, for barriers.
- Transfer counters (TransferCtrs) can be configured to count the data that messages of a specific handler code have written into memory. They also have a configurable threshold. This is useful for *soft* barriers, where 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 programming benchmarks.
- The timer can be modified by a particular offset, which is useful when multiple FPGAs did not come out of reset in the same cycle.
- When a message has arrived, the current timer is copied into the *ArrivalTime* register. The value can be polled by sending a special message whose handler code does not trigger the copying.
- The sequencer can set control outputs to the custom hardware and wait on control inputs from the custom hardware.
- 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 code, from the timer or from the *ArrivalTime* register.

Figure 6 shows pseudocode for implementing a simple barrier in a 16-node system, as benchmarked in Section 5.2. There are two different code versions, one 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.

All benchmarks in this paper were realized through corresponding configuration of the sequencers in each core. The PAMS could also be used to implement the GASNet Extended API mentioned in Section 2.

```
/* 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);

/* node 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 6: Example PAMS code

Table 2: PAMS synthesis statistics, Xilinx XC5VLX155T-1

Resource	#	%
Registers	943	0.97
LUTs	1035	1.06
BRAM18s	1	0.24

Table 2 shows the resource utilization at a maximum clock speed of 136.2MHz, projected by synthesis for a PAMS with four control inputs and four control outputs on a Xilinx XC5VLX155T-1. As was the case with the GAScore, we can see that a single PAMS uses a very small amount of logic resources.

3.3 On- and Off-chip Networking

A system of several GAScore-equipped computing nodes on one FPGA is pictured in Figure 7. In this example, two nodes use an embedded processor and two nodes use a hardware processing element (PE). Each GAScore connects to an on-chip network of *NetIfs*. NetIfs are simple FSL-based cut-through routers originally introduced for use by an on-chip MPI system in [21]. The NetIfs are arranged in a fully connected network. The feasibility of such fully-connected networks with FPGA routing resources has been examined in [22]; however, other topologies are under consideration for larger FPGAs where routing fabric does not increase proportionately with logic area.

Off-chip connections through other network and peripheral interfaces can be implemented through bridge components that can connect to NetIfs. In Figure 7, two Off-Chip Communication Controllers (OCCC) manage external data transfer to two different directions. Depending on the OCCC, communication can, for example, happen over board-level connections, optical or copper-based networks or PCI host buses. The choice of external interface does not influence GAScore functionality since its communication model is implemented on top of the physical network layer.

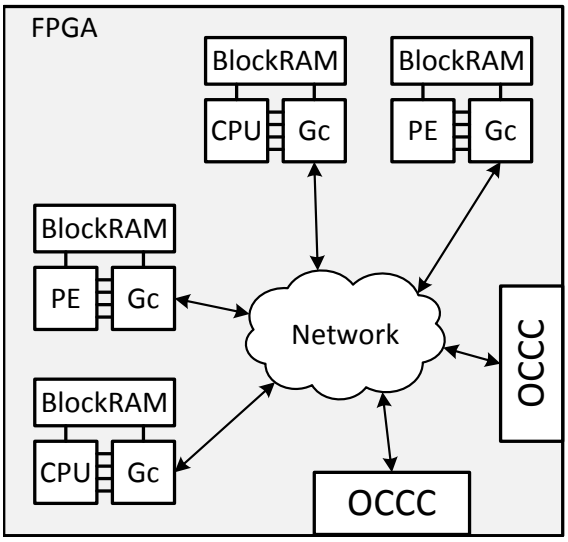

Figure 7: Four-node system with Off-Chip Communication Controllers

3.4 Portability

As explained in Section 3.2, GASNet software only needs a lightweight interface layer to control the GAScore component because its Active Message requests and Handler calls use the same parameters as the corresponding C functions. For the same reason, FPGA communication with host architectures running GASNet will be straightforward. The challenge is reduced to implementing bridges between host hardware and FPGA NetIfs, something which has already been successfully demonstrated in previous work[21].

A further advantage of this compatibility is the possibility to prototype code on a host architecture and then easily migrate it to an embedded processor. When CPU computation is translated into a pure hardware implementation at some point, the patterns of data movement and synchronization can be easily preserved.

4. SOFTWARE STACK AND CODE GENERATION PROCESS

Figure 8 illustrates the software stack that we envision on top of our component infrastructure. GASNet serves as the basic communication layer between all components. On top of the GASNet library sits a C++ library currently under development that unites proven PGAS and heterogeneity concepts from existing languages and libraries. Its main features are:

- Complex data classes for multi-dimensional arrays, etc.
- Location and node subset classes that allow modeling of heterogeneous systems
- Data layout types to control platform-specific data distribution independently from the data class itself

Applications can be written in C++ to run with the PGAS library. However, many scientists use Domain-Specific Languages (DSLs) that enable more productive and efficient modeling of problems in their specific area. We envision our C++ library to also be a suitable runtime environment

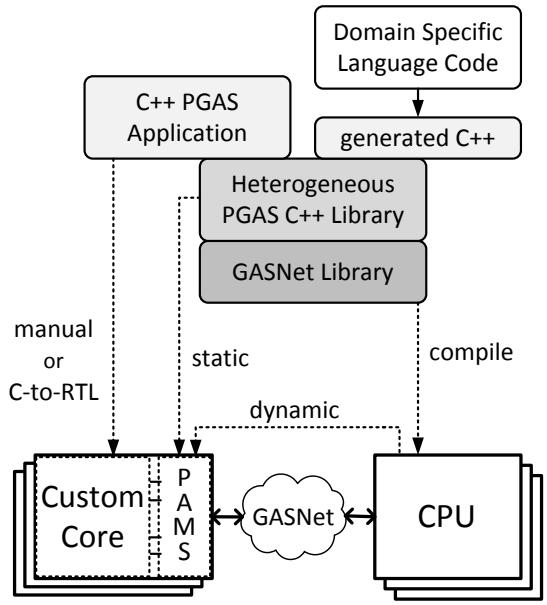

Figure 8: PGAS software stack and code/binary generation

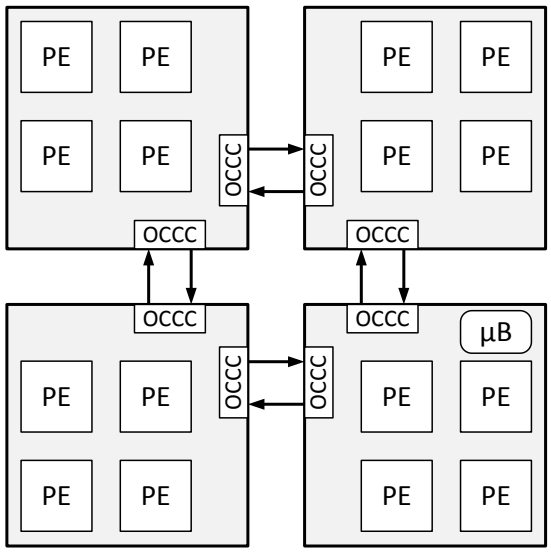

Figure 9: BEE3 quad-FPGA system with 16 hardware processing elements (PE) and one MicroBlaze processor(μB) for configuration and benchmarking purposes

for C++ code that was automatically generated from DSL code.

For debugging purposes, an application should always be able to execute correctly (though possibly inefficiently) after compilation to a pure multi-CPU environment running GASNet. Based on profiling, the application designer can decide which functions should be migrated to hardware. Performance-critical functions can then be translated into FPGA hardware, either manually by a skilled hardware designer or through High-Level Synthesis (HLS/"C-to-gates") tools.

The communication and synchronization patterns that the PAMS needs to run for the new hardware kernel can be generated automatically by the library, either statically at compile time or, if required, dynamically at runtime. The ability to re-program the PAMS through GASNet enables dynamically changed control patterns at any time.

Since our C++ library is still in a very early design stage, all evaluations and benchmarks of our hardware components so far have been done just with the GASNet library and hand-written PAMS code.

5. PERFORMANCE EVALUATION

For a parallel computation system, it is imperative to keep the cost of remote data accesses as low as possible, so that exchange of data with other computation nodes does not carry an excessive penalty. Therefore, our main concern in evaluating our component is that remote data can be read and written with relatively low latency. Furthermore, we measure the maximum latency of two types of barriers, since barrier latency is performance-critical for many real-world applications. Given foreseeable contention for shared networking resources, a second aspect we investigate is how much overhead single data transfers incur, and therefore how efficiently the available network bandwidth can be used.

We have deliberately not included any application benchmarks in our evaluation. At this stage of development, it is important to understand the exact performance of this first implementation of our memory-to-memory transfer core using the NetIf infrastructure. Any application benchmark would be defined by the quality of a custom hardware core implementation, as well as the network bandwidth and congestion that is characteristic to this specific topology and board hardware (In fact, these characteristics *do* strongly affect our barrier benchmarks). We would not be able to draw definitive conclusions on whether our component or another influence is the impediment to better performance.

5.1 Test system

As a testbed, a demonstration system is implemented on a BEE3[16] multi-FPGA board. The BEE3 holds four Xilinx XC5VLX155T FPGAs. They are connected in a bi-directional ring with 32-bit-wide communication at 100MT/s in each direction. The system is clocked at 100MHz.

A single FPGA holds four hardware computation nodes and one Microblaze processor, each connected to 64Kbytes of BlockRAM and one GAScore. All four GAScores, as well as two off-chip communication controllers for both ring directions, are connected to one NetIf each, resulting in a fully-connected topology of seven NetIfs as illustrated in Figure 7. Figure 9 shows the complete system, with the four FPGAs building a network of 17 computation nodes. Nodes 0-15 are identical hardware processing elements (PE). Node 16 is the Microblaze processor (μB) in the fourth FPGA; the other three Microblaze processors are not utilized in the system. The Microblaze is used to program the PAMS in each processing element, poll the other nodes for any benchmark results and output the results through a RS232 UART.

Table 3 shows the implementation results for one chip with four hardware nodes and one MicroBlaze processor, using Xilinx Platform Studio 10.1.03.

Table 3: Test system implementation results, Xilinx XC5VLX155T-1

Resource	#	%
Registers	11160	11
LUTs	18651	19
Slices	8058	33
BRAMs	98	46

Table 4: Short message latencies between nodes

Ring distance	1-way (us)	2-way (us)
0	0.17	0.35
1	0.24	0.49
2	0.31	0.63

Table 5: Single-word memory transfer latencies

Ring distance	1-way (us) remote write	2-way (us) remote read
0	0.29	0.47
1	0.36	0.61
2	0.43	0.75

The system could be run at up to 138MHz with the same resource use. Clearly a decidedly larger system could be implemented on each FPGA. However, because of the fully connected topology, routing resources are expected to run out before logic does.

5.2 Latency results

Initially, we tested how long a single short message between two nodes takes from sending to receive completion. Because nodes inside an FPGA are fully connected and we tested without any congestion, results do not differ depending on which node inside the same FPGA was used for a measurement. This applies to all congestion-free measurements. Consequently, latency only differs with varying off-chip distances, which is why for a 4-chip ring we get values for 0, 1 and 2 FPGA hops. The latency for a short message varies between 170ns for an on-chip message to 310ns for a 2-hop message, equivalent to 17 and 31 clock cycles. A 2-way or "ping-pong" message takes only slightly more than double the time, since our PAMS allows very quick turnaround.

Next, we determined how long the smallest possible transfer from memory to memory takes, equivalent to remotely writing a datum. Latency increases by 12 cycles or 120us. This is only partially due to memory access latency: To ensure correctness, the GAScore only sends the handler call to the computing element when the memory has been completely written. This is in contrast to short messages, where parameters are transmitted onwards in cut-through style before the network packet has completely arrived. The latency for the equivalent of a remote read can be easily predicted since this is a combination of a short request and a long reply, and therefore only adds the previous 12-cycle delay to one side of the original 2-way latency.

For barrier latency, we first evaluated a straightforward implementation where all nodes send a barrier request to node 0; after receiving all 15 requests, node 0 sends *barrier_done* signals back to all other nodes. Table 6 shows how long it takes for all messages to reach node 0 (which is when the first node is done with the barrier) and how long it takes for all nodes to be notified about the completed barrier.

Table 6: Simple barrier latency (us)

Latency to node 0	0.87 us
Latency for all	1.96 us

Table 7: "Staggered" barrier latency (us)

Latency to node 0	0.62 us
Latency for all	1.48 us

Contrary to the previous measurements, for the barrier the off-chip connections become a bottleneck where several messages contend for the same channel that can only transmit one message at a time.

To alleviate the bottleneck, our second measurement uses a staggered or tree-based barrier: Every node sends their barrier request to an on-chip node that functions as a hub. Only those hub nodes connect off-chip to node 0. Node 0 sends the *barrier_done* message back to the hub nodes, who distribute it to their on-chip neighbors. As all presented benchmarks, this change is implemented just through reprogramming the PAMS in each node, no hardware changes are necessary.

Table 7 shows the results. They are not dramatically better for two reasons: First, the three hub requests to node 0 from off-chip contend with the three local requests on the first chip, for which node 0 is the hub. Node 0 therefore still receives six of the previous 15 requests. Secondly, the off-chip non-hub nodes now have a longer communication latency for each single request, since they have to go through two nodes instead of one.

5.3 Bandwidth results

We further examined the transfer times for memory-to-memory operations of different sizes to determine how big the impact of latency is on effective bandwidth. Figure 10 shows that we reach about half of the optimum bandwidth at transfer sizes between 64 and 256 bytes, but that bandwidth for smaller transfer sizes suffers decidedly. These results are problematic, since PGAS enables user applications to commonly read and write single words of remote data, something that is heavily penalized here in terms of throughput. Almost any network infrastructure allows network saturation on large data packets, however ours clearly needs improvement on small amounts of data.

5.4 Discussion

It is clear from our latency numbers that a custom implementation of a memory-to-memory transfer could do in a few cycles what our communication infrastructure does in 17 or more cycles. Part of this disadvantage is a trade-off for a programming model that is easier to manage. This becomes clear as soon as the remote memory access happens across the board to another chip: Considerable effort is necessary to integrate a chip-to-chip interface into the communication. To the GAScore user, communication to an on-chip neighbor or any off-chip location is completely identical and does

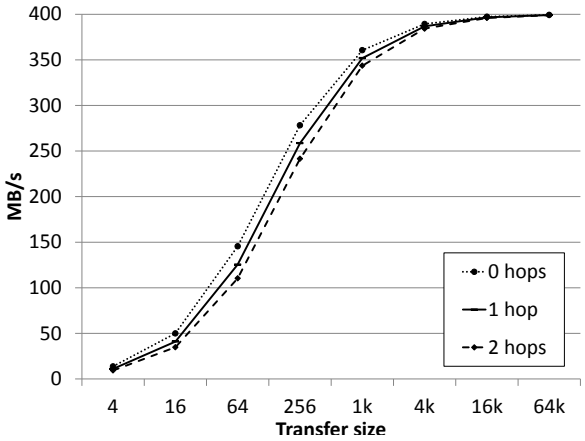

Figure 10: Effective bandwidth depending on transfer size

not introduce additional complexity. Furthermore, systems of dozens or hundreds of nodes stay easily manageable.

However, there is certainly room for improvement in the GAScore performance. A major bottleneck is introduced by serializing Active Message requests and Handler calls through the 32-bit FSLs. This is unavoidable when using a MicroBlaze processor, however a custom hardware core could quite well transmit all the parameters in parallel. In fact, we are considering moving the complete PAMS into the GAScore to minimize the delay in receiving, processing and sending Active Messages. The GAScore would in that case only connect through control bits to custom hardware cores. We will examine if this approach would introduce other disadvantages. Similarly, latency as well as bandwidth on the NetIf side could be improved significantly by extending from a 32-bit to a 128-bit network, something that TMD-MPI has already done.

A positive side effect of our benchmarks has been to demonstrate how the introduction of the Programmable Active Message Sequencer has helped testing productivity. In our early attempts at evaluation, every benchmark had to be hardcoded into custom logic and resulted in another place-and-route run. With the PAMS, small modifications and experiments with different sequences (e.g. the second barrier benchmark) can be implemented with a simple bitstream update. We are planning to further ease development of communication patterns by writing a PAMS/GASNet emulator.

6. RELATED WORK

Substantive prior work has been done to connect on-chip memory resources more smoothly with each other and with off-chip resources. CoRAM as introduced by Chung et al.[15] 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.

The approach of having a similar, migration-conducive API for software and hardware components has been successfully used by Saldana et al.[21]. 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 model for the simulation of molecular dynamics. Its 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, finally, the impediments to dynamic memory accesses and the use of linked data structures.

Hthreads [10] 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.

VForce [19] extends VSIPL++, an established C++ high-performance computing and signal processing library, to use reconfigurable hardware for select library functions. VForce supports runtime binding so that the same application code can run on systems with and without supporting reconfigurable hardware. We expect our heterogeneous PGAS C++ library to behave in a similar fashion, however with a focus on PGAS coding concepts and (optionally) more explicit awareness of heterogeneity for the application programmer. Furthermore, VForce seems to focus on heterogeneity in a single node, while scalability across larger systems and networks is integral to our architecture.

On the PGAS side, SHMEM+ [9] 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. The SHMEM+ authors exclude on-chip memory from being remotely accessible, concluding that it is only useful for caching purposes. We think that there are classes of applications with small, latency-critical datasets, and therefore provide this access, especially since the exchangable memory component in our GAScore means there is no added cost to it.

El-Ghazawi et al.[18] 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 *upc_forall*-statements that can be well parallelized in hardware and splits their

compilation off to Impulse-C[2]; corresponding data transfers to and from the FPGA are inserted into the CPU code.

The 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, and FPGA capabilities are underutilized.

7. FUTURE WORK

Our most urgent work lies in improving latency as discussed in section 5.4. We are optimistic that we can improve performance significantly with the suggested changes.

In the short term, we plan to extend the GAScore memory interface to off-chip DRAM to open the system to a larger set of applications and data set sizes. A common multi-ported memory controller supports eight ports, so that four processing nodes with one processing element and one GAScore each could access a DRAM module.

For many applications on distributed arrays, built-in strided and scatter-gather accesses would be beneficial and could minimize the workload portion that a processing element needs to spent on initiating data transfers.

Our long term intentions are focused on the software stack described in Section 4. GAScore and the Programmable Active Message Sequencer are laying the groundwork for this by creating a common interface for processors and hardware cores. We plan to examine how to best model heterogeneity in a high-level language PGAS implementation and how to best migrate performance-critical kernels into logic.

8. CONCLUSION

We introduced a remote memory communication engine, GAScore, which can be easily interfaced by embedded processors as well as hardware engines on FPGAs. Furthermore, we introduced PAMS, a communication controller that simplifies using custom hardware cores with GASNet. Compatibility to a popular shared memory networking library, GASNet, assures easy integration with host-based parallel programming solutions and enables development of heterogeneous computing applications. We discussed how these low-level components fit into a larger approach for computing application development.

Our evaluation and results have shown us two things: First, there is room for improvement in the achieved latency for messages and data transfers. We have several ideas on how to optimize performance and improve those results. Secondly, we succeeded in providing an easy-to-use set of components for remote memory access. Especially the flexibility added through the Programmable Active Message Sequencer facilitates easier integration of custom hardware into a shared-memory-based parallel processing system. The ability to change communication and control patterns without a complete implementation run boosts design productivity. Like GAScore, PAMS delivers its functionality on a very low area budget.

At this point, our components provide the envisioned low-level functionality, but for a competitive system we need to markedly improve performance. Furthermore, we need to add the necessary software and hardware components to extend the existing infrastructure into a truly productive, heterogeneous, parallel programming environment.

9. ACKNOWLEDGEMENTS

We thank the CMC, NSERC and Xilinx for supporting our research. Special thanks to Manuel Saldaña of Arches Computing Systems for his BEE3 compile scripts and the NetIf infrastructure.

10. REFERENCES

[1] Co-Array Fortran. http://www.co-array.org/.

[2] Impulse C. http://www.impulseaccelerated.com/.

[3] Infiniband trade association. http://www.infinibandta.org/.

[4] Message passing interface forum. http://www.mpi-forum.org/.

[5] OpenMP application programming interface version 3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[6] OpenSHMEM application programming interface. http://openshmem.org/.

[7] Titanium. http://titanium.cs.berkeley.edu/.

[8] Unified parallel C. http://upc.gwu.org/.

[9] V. Aggarwal, A. D. George, C. Yoon, K. Yalamanchili, and H. Lam. SHMEM+: A multilevel-PGAS programming model for reconfigurable supercomputing. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):26:1–26:24, Aug. 2011.

[10] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 89 –98, april 2006.

[11] D. Bonachea and J. Duell. Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Perform. Comput. Netw.*, 1(1-3):91–99, Aug. 2004.

[12] D. Bonachea and J. Jeong. GASNet: A portable high-performance communication layer for global address-space languages. Cs258 parallel computer architecture project report, University of California Berkeley, Spring 2002.

[13] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 04)*, pages 52–60, 2004.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[15] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[16] J. Davis, C. Thacker, and C. Chang. BEE3:

Revitalizing computer architecture research. Technical report msr-tr-2009-45, Microsoft Research, April 2009.

[17] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. McMahon, A. Snavely, J. Vetter, K. Yelick, S. R. Alam, R. Campbell, L. Carrington, T.-Y. Chen, O. Khalili, J. S. Meredith, and M. Tikir. *DARPA's HPCS Program: History, Models, Tools, Languages*, volume Volume 72, pages 1–100. Elsevier, 2008.

[18] T. El-Ghazawi, O. Serres, S. Bahra, M. Huang, and E. El-Araby. Parallel programming of high-performance reconfigurable computing systems with Unified Parallel C. In *Proceedings of Reconfigurable Systems Summer Institute*, 2008.

[19] N. Moore, A. Conti, M. Leeser, and L. King. Writing portable applications that dynamically bind at run time to reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 229 –238, april 2007.

[20] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: a portable *shared-memory* programming model for distributed memory computers. In *Supercomputing '94. Proceedings*, pages 340 –349, 816, nov 1994.

[21] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam. MPI as a programming model for high-performance reconfigurable computers. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):22:1–22:29, Nov. 2010.

[22] M. Saldaña, L. Shannon, and P. Chow. The routability of multiprocessor network topologies in FPGAs. In *Proceedings of the 2006 International Workshop on System-level Interconnect Prediction*, SLIP '06, pages 49–56, New York, NY, USA, 2006. ACM.

[23] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.