CoRAM++: Supporting Data-Structure-Specific Memory Interfaces in FPGA Computing

Gabriel Leonard Weisz

CMU-CS-15-135
September 2015

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
James C. Hoe (Chair)
Kayvon Fatahalian
Kenneth Mai
Todd Mowry
Joel S. Emer (M.I.T. / NVIDIA)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2015 Gabriel Leonard Weisz

This research was sponsored by the National Science Foundation under grant number CCF-1012851, the Industrial Technology Research Institute under grant number ITRIMSL2008, and Intel.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Computer Architecture, Reconfigurable Computing, FPGA Computing, Decoupled Computing, Development Environments, FPGA, Hardware Specialization, Data-Structure Libraries, FPGA Abstraction, FPGA Design Methodology

For my son Zach. I hope that you never lose your joy in exploring the world.

Abstract

FPGAs offer high performance and power efficient computation, but are difficult to use. In particular, the effort involved in managing data movements between on-chip computation components and off-chip DRAM has prevented FPGAs from being widely adopted by the computing industry. Recently developed FPGA programming environments layer simplifying abstractions on top of the DRAM interfaces provided by FPGA vendors, but existing programming environments have primarily focused on support for simple, regular data access patterns such as block copy and streaming.

This thesis proposes CoRAM++, an FPGA programming environment that efficiently supports complex data structures such as multi-dimensional arrays and linked lists in addition to simple data access patterns. CoRAM++ application developers manage data movements through an extensible library of data-structure-specific application-level interfaces, which generate specialized soft-logic datapaths between application components and memory. This extensible library of data-structure-specific application-level interfaces is layered top of a system interface which allows library components to attach modules directly to a memory interface in order to lower the latency of irregular, pointer chasing operations. Our evaluation of CoRAM++ shows that this approach can provide convenient data access to a variety of data structures without introducing undue performance or resource overheads, which should make CoRAM++ attractive to FPGA application developers.

Acknowledgments

Thank you James Hoe, for supporting me, guiding towards interesting work, and encouraging me to look into anything that might be useful. You have an incredible ability to discern which aspects of a project are most interesting, and to recruit great students and post-docs.

Thank you committee members: Ken Mai, Kayvon Fatahalian, Todd Mowry, and especially Joel Emer, my external committee member who trekked from Boston to Pittsburgh for my thesis defense. All of you helped me make sure the that the work went in the right direction.

Thank you Eric Chung and Michael Papamichael for immeasurable help. Thank you to all of James' students who concurrent with me at CMU: Peter Milder, Peter Klemperer, Berkin Akin, Yu Wang, Marie Nguyen, Joe Melber, Zhipeng Zhao. Thank you to my GraphGen collaborators: Eriko Nurvitadhi, Skand Hurkat, José Martínez, Carlos Guestrin, for a good project. Thank you to all of CALCM for good conversations and good talks. Thank you A-levelers for a making it a good place to work, and for coming out to lunch. Thank you to all of my friends at CMU, particularly the 7th floor and A-level crews.

Thank you to everyone in my family. Thank you to my parents for stressing the importance of education and hard work, and for supporting me in everything I've ever done. Thank you to my grandparents, for your bravery in leaving Romania to build a life in the U.S. and demonstrating that the American dream still exists for those who are willing to work for it. Thank you Dave, the first to earn a PhD, for showing me the way. Thank you Jon, for making it clear that you need both persistence and chutzpah to finish a PhD. Thank

you Cat for everything good in my life. Without you I wouldn't have seen how much fun pursuing a PhD was. You give me inspiration and more support than I could imagine.

Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**

1.1 Enabling Data-Structure-Specific Memory Interfaces 3

1.1.1 A Multi-Level System Architecture 3

1.1.2 Memory Interfaces Composed of Control and Data Sub-Interfaces 5

1.2 Decoupled Computing and the Original CoRAM Architecture 7

1.3 CoRAM++ Data-Structure-Specific Interfaces 8

1.4 Thesis Contributions 9

1.5 Thesis Outline 10

2 Background and Related Work **13**

2.1 Computing with FPGAs 13

2.2 Soft-logic and Hard-logic Processor Cores on FPGAs 14

2.3 Higher Level Hardware Description Languages 16

2.4 High-Level Synthesis Tools 17

2.5 Bus infrastructures and Networks-on-Chip 18

2.6 FPGA Computing Application Programming Environments 19
2.7 The CoRAM Architecture for FPGA Computing 21
2.7.1 Mapping the CoRAM Architecture to Soft-Logic FPGAs 22
2.7.2 C-to-CoRAM: CoRAM as a High Level Synthesis Target 24
2.7.3 Limitations of the CoRAM Architecture and its API 27
3 The CoRAM++ Programming Environment for FPGA Computing 31
3.1 Arguments for Specializing in a Soft-Logic Context 31
3.2 Complexities in Creating Extensible Interfaces 33
3.3 CoRAM++ Scope and Underlying Assumptions 37
3.4 CoRAM++ Programming Environment Overview 39
3.5 Example CoRAM++ Application 41
4 Currently Supported Application-Level Interfaces 45
4.1 Data-Structure-Specific Interfaces for Memory Access Patterns 45
4.1.1 Block Copy Interface 46
4.1.2 Stream Interface 47
4.1.3 Multi-dimensional Array Interface 48
4.1.4 Linked List Interface 52
4.1.5 B-Tree Interface 53
4.1.6 Gather-Scatter Work-List Interface 55
4.2 Utility Interfaces 57
4.2.1 Scratchpad Interface 57
4.2.2 Host Computer Interface 59

4.2.3 Channel Interface 60

4.2.4 Kernel-Kernel Communication Interface 61

4.2.5 Thread Creation Interface 62

5 Inside the CoRAM++ Programming Environment **63**

5.1 The CoRAM++ Compiler 63

5.2 CoRAM++ Control Thread Capabilities and Limitations 68

5.3 CoRAM++ Built-in Interfaces 70

5.3.1 Block Copy Interface 70

5.3.2 Channel Interface 71

5.3.3 Thread Creation Interface 71

5.3.4 Script Execution Interface 71

5.3.5 Control Thread System Library Interface 72

5.4 CoRAM++ Accelerator Agents and Software Accelerators 72

5.5 CoRAM++ Cache 74

6 Constructing the CoRAM++ Agents **79**

6.1 Stream Agent 80

6.2 Multi-Dimensional Array Agent 81

6.3 Linked List Agent 81

6.4 B-Tree Agent 85

6.5 Gather-Scatter Work-List Agent 85

6.6 Scratchpad Agent 87

6.7 Host Computer Agent 88

6.8 Kernel-Kernel Communication Agent 92

7 Evaluating the CoRAM++ Programming Environment 95

7.1 Streaming Accesses 95

7.2 Multi-Dimensional Array Accesses 97

7.2.1 Strided Array Accesses 98

7.2.2 Tiled Array Accesses 98

7.3 Pointer-Chasing Linked List Accesses 101

7.3.1 Linked List Traversal and Merging in Soft Logic 101

7.3.2 Linked List Traversal using an ARM Processor Core 105

7.4 Graph Computations Using the Gather-Scatter Work-list Agent 121

7.4.1 Depth Reconstruction 124

7.4.2 Handwriting Recognition 127

7.5 Streaming Data Compression 128

7.6 Kernel-Kernel Communication 129

8 Conclusion 133

Appendix A Application Notes for Supported Compiler Targets 137

A.1 General Application Notes 137

A.2 ML605 Compiler Target Notes 138

A.3 DE4 Compiler Target Notes 139

A.4 ZedBoard and ZC706 Compiler Target Notes 140

Appendix B Host Computer Agent Communication Details 145

B.1 Host Computer Interface Communication Protocol 147

B.2 Standard Host Computer Interface Client Program 147

B.3 Custom Client Program for the Handwriting Recognition Demo 148

References **151**

List of Figures

1.1 CoRAM++ system architecture 4

1.2 A memory interface composed of control and data sub-interfaces 5

1.3 CoRAM/CoRAM++ application decomposition 6

2.1 Simulation results using the CoRAM Architecture. 22

2.2 C-to-CoRAM compilation workflow. 25

3.1 Application-level view of a streaming CoRAM++ DFT application. 42

4.1 Tiled data layout 51

4.2 B-Tree structure as compared to typical binary search trees 54

5.1 System-level view of a complete CoRAM++ application 64

5.2 Accelerator Agent Wrapper Module 73

6.1 Stream agent diagram 80

6.2 Mult-dimensional array agent diagram 82

6.3 Linked list engine. 83

6.4 Gather-scatter work-list agent diagram 86

6.5 Scratchpad agent diagram 88

6.6 Host computer agent diagrams for FPGAs without processor cores 89

6.7 Host computer agent diagrams for FPGAs with processor cores 90
6.8 Kernel-kernel communication agent diagram 93

7.1 Saturating DRAM interfaces 96
7.2 2D and 3D DFT Performance 100
7.3 Linked list configurations 103
7.4 Linked list traversal performance 103
7.5 Sorted linked list merge performance 105
7.6 Datapaths between memory, the ARM cores, and the fabric in the Zynq . 106
7.7 Performance achieved on simple linked lists 109
7.8 Performance achieved with larger indirect data payloads 112
7.9 Performance achieved with interleaved memory accesses 115
7.10 Vertex-centric graph description. 122
7.11 GraphGen compiler workflow. 123
7.12 Depth Reconstruction Application Configuration and Performance 124
7.13 Handwriting Recognition Application Configuration and Performance . . 126
7.14 Compression kernel performance. 128
7.15 Kernel-Kernel Communication Agent Performance Results. 130
7.16 Kernel-Kernel Communication Application Resource Utilization. 131

B.1 Screen capture of generic host communication program 147
B.2 Screen capture of host communication program for handwriting recognition 149

List of Tables

4.1 Block copy application-level interface. 46
4.2 Read stream application-level interface. 47
4.3 Write stream application-level interface 48
4.4 Read multi-dimensional array application-level interface. 49
4.5 Write multi-dimensional array application-level interface. 50
4.6 Linked list application-level interface. 53
4.7 B-tree application-level interface. 55
4.8 Hardware ports provided by the gather-scatter interface 56
4.9 Scratchpad application-level interface for control threads. 58
4.10 Host computer application-level interface for control threads. 59
4.11 Channel application-level interface. 60
4.12 Kernel-kernel communication interface for control threads 61

5.1 Supported FPGA boards. 66
5.2 CoRAM++ system library for control threads 76
5.3 Interface that accelerator agents must support. 77

7.1 CoRAM++ 1D DFT resource breakdown by component. 97
7.2 Resource utilization for CoRAM++ and reference DFTs. 100

B.1 Host computer interface message definitions. 146

Chapter 1

Introduction

2015 marks the 50th anniversary of Moore's Law [63]. Moore's Law is an observation that the number of transistors in mass-marked computer chips doubles on a regular schedule due to decreasing transistor size. However, this schedule has just slipped from 2 years to 2.5 years [42], and Moore's Law is not the whole story for performance—clock frequencies have been stagnant for the last 10 years [2]. As Moore's Law has slowed down and clock frequencies have peaked, the high performance computing community first turned to parallel computing to obtain performance from increased transistor counts, and then created heterogeneous computing systems that combine many CPUs and GPUs into massively parallel supercomputers [5].

Field Programmable Gate Arrays (FPGAs) can further improve performance and reduce power requirements [26, 30, 40, 78]. The computing industry's largest players are working to bring FPGAs into the data center [23, 74], demonstrating real-world interest in harnessing the potential of FPGAs. However, it remains difficult to create FPGA

computing applications, both because of the effort involved in mapping computations to computation kernels, and because of the effort involved in supplying these computation kernels with data. High level synthesis tools and IP core generators (see Chapter 2) make it easier to map computation to the FPGA fabric by generating computation kernels from software source code, and recent work on FPGA programming environments makes it easier to manage data. But existing FPGA programming environments focus primarily on simple, regular data access patterns such as block copy and streaming, forcing the application developer to implement more complex behaviors that may be needed by the application.

This thesis presents CoRAM++, which extends efficient support to more complex data structures, such as arrays, linked lists, and graphs. CoRAM++ simplifies FPGA application development through an extensible library of portable data-structure specific memory interfaces that are easy to use and perform well. These memory interfaces allow applications to trigger data transfers by issuing simple, data-structure-specific commands through a software programming model. For example, a memory interface for a linear array would allow the application to trigger streaming data transfers to or from memory, while a memory interface for a multi-dimensional array would allow the application developer to stream blocks of data across any dimension. An interface for a pointer-based data structure would allow the application to traverse the data structure, and might also provide support for modifying pointers in order to change the relationships between data items. Use of CoRAM++ memory interfaces does not necessarily incur a performance penalty—these interfaces can match the best performance achievable when supporting sequential data accesses, and minimize the latency incurred when supporting irregular, pointer chasing data accesses.

CoRAM++ supports an extensible set of regular and irregular data access patterns, and allows application developers to create the application once and run it on a variety of FPGA boards from different manufacturers by simply recompiling the application, rather than manually creating new logic for each board's unique DRAM and communication interfaces. CoRAM++ targets a decoupled computing paradigm that separates an application into computation and communication components, enforcing a separation of concerns between computation and communication needs, and enabling a software-like programming model for managing communication. This decoupled computing paradigm is a good fit for the way application developers commonly build FPGA applications, which are often designed around computation kernels selected from IP libraries or generated by high level synthesis tools.

1.1 Enabling Data-Structure-Specific Memory Interfaces

CoRAM++ incorporates two key architectural features that allow the CoRAM++ programming environment to efficiently and conveniently support data-structure-specific memory interfaces:

- A multi-level system architecture composed of a low-level system layer and an extensible high-level library layer.
- Memory interfaces within the library layer are composed of separate sub-interfaces for control and data.

1.1.1 A Multi-Level System Architecture

CoRAM++ defines a multi-level system architecture in which application components interact with an extensible library layer providing application-level data-structure-specific

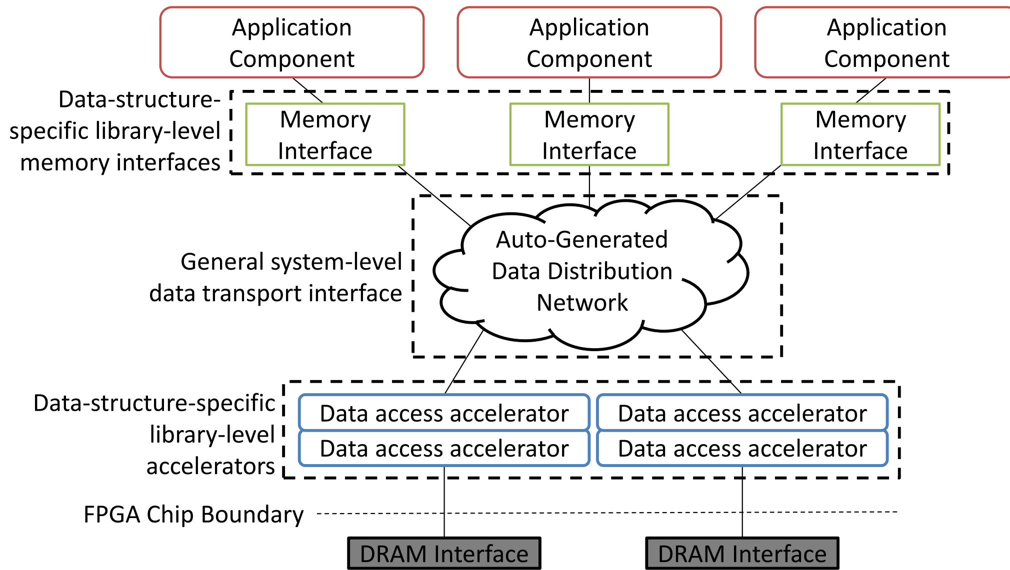


Figure 1.1: CoRAM++ system architecture including a high-level library layer that interacts with a low-level system layer, and can connect data access accelerators directly to a DRAM interface.

interfaces, which is mapped onto a system that providing low-level services. Figure 1.1 depicts this system architecture, showing application components, the library layer, the system layer, and data access accelerator modules that library components can instantiate.

The system layer manages with DRAM and other off-chip communication interfaces and on-chip data transport. The system layer provides a low-level, generic interface that is as simple as possible, in order to ensure an efficient and correct implementation. The extensible library layer builds convenient data-structure-specific interfaces on top of the system layer, and can instantiate data access accelerator modules at DRAM interfaces for performance reasons. These accelerator modules can be hardware engines that are instantiated within the reconfigurable fabric, or can be software running on a tightly integrated hard-logic processor core. The application is not aware of the implementation of the li-

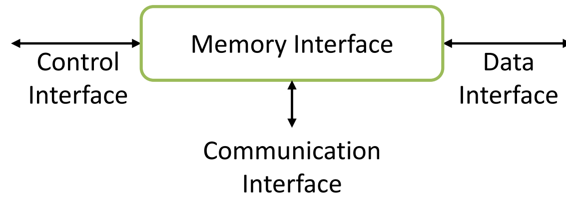


Figure 1.2: A data-structure-specific application-level memory interface composed of control and data sub-interfaces, and including a communication interface to the NoC.

brary components in use, or the existence of accelerator modules, but instead accesses the selected data-structure-specific interfaces through a data-structure-appropriate commands.

This multi-level application development environment is analogous to software development. Software development environments generally include an operating system that supports low-level functionality such as memory allocation and hardware access, and libraries that support high-level functionality, such as formatted I/O and standard implementations of common data structures. Data access accelerators are somewhat similar to the functionaly provided by CUDA [68]—acceleration of particular types of operations on specialized hardware. In the case of CUDA, this means SIMD-style code that runs well on a GPU, and in the case of CoRAM++ data access accelerators, this means particular data access functions that run well on specialized hardware attached directly (or in a short pipeline) to a memory interface to minimize latency to that interface.

1.1.2 Memory Interfaces Composed of Control and Data Sub-Interfaces

Figure 1.1 presents the system architecture above used by CoRAM++ to support data-structure-specific memory interfaces. Each of these memory interface includes three sub-interfaces, which are depicted in Figure 1.2:

1. A control interface that is used to send commands to the interface, which can trigger

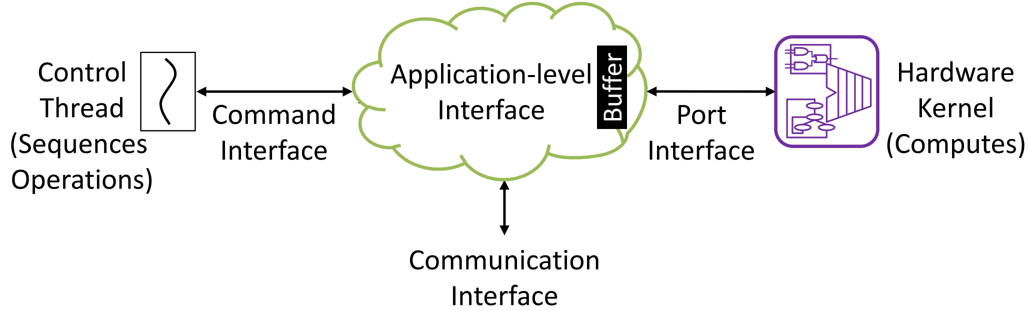


Figure 1.3: CoRAM/CoRAM++ application decomposition into separate compute and control.

or monitor the status of data transfers. The commands supported by each interface are specific to the type of data structure managed by the interface,

2. A data interface which provides data for computation.

3. A communication interface which connects to the data-distribution network.

Applications consist of modules that attach to the control and data sub-interfaces of the selected memory interfaces. This approach does not define the specific commands that go across the control interface or the ports provided by the data interface (such as SRAM-style or FIFO-style ports) supported by each memory interface, but does define that there are separate control and data sub-interfaces, This approach does not preclude a single module from attaching to both the control and data sub-interfaces of a memory interface. However, CoRAM++ defines a decoupled programming model does define how application components interact with memory interfaces.

1.2 Decoupled Computing and the Original CoRAM Architecture

CoRAM++ follows a decoupled computing paradigm [77] that separates applications into modules that sequence operations and modules that perform computation. Figure 1.3 shows this application decomposition, which attaches the different types of modules to the data and control interfaces described in Section 1.1.2. Hardware kernels perform computation, interacting only with locally buffered data. Control threads sequence data transfers and the invocation of hardware computation kernels.

CoRAM++ uses a programming model developed for the original CoRAM architecture [31], which defines that control threads are written in a multithreaded C-like language and compiled to state machines. This decoupled computing model shields computation kernels from the complexity of hardware platform-specific interactions with external DRAM, and shield the application developer from the need to manually create state machines. Since control threads manage long running operations, they do not need the cycle-level control that hardware description languages provide, and the fact that they are compiled from software code (and may not be as optimized as hand-designed state machines) does not hurt performance.

The application-level interfaces used by CoRAM and CoRAM++ applications are divided into two distinct sub-interfaces, as described above in Section 1.1.2:

1. Kernels access data through wire-level port interfaces.
2. Control threads issue commands through function call interfaces.

The original CoRAM architecture natively supports a block copy application-level interface for DRAM access:

1. Kernels access data through wire-level SRAM port interfaces.
2. Control threads issue commands (by making function calls) to transfer data between external DRAM and local SRAM blocks.

CoRAM's application-level interface was meant to be hardened into a future FPGA, where it would serve as a universal primitive used to support all memory access patterns. In addition to DRAM interfaces, the original CoRAM architecture supports communication interfaces such as PCI express, ethernet, and serial ports, which are mapped into a global address space and accessed using the natively supported block copy application-level interface. CoRAM++ uses a soft-logic mapping of the CoRAM architecture as its system-level interface, which was enhanced with features that enable CoRAM++ to support convenient. and is described in Section 5.3.

1.3 CoRAM++ Data-Structure-Specific Application-Level Interfaces

CoRAM's application-level interface can be realized in the soft logic of today's FPGAs, but with penalties in terms of both performance and logic overhead [32]. This application-level interface also limits the ways in which libraries can layer new functionality on top of its native application-level interface, as will be illustrated by Section 2.7.3. CoRAM++ resolves these issues by by making better use of the reconfigurability of the FPGA fabric to provide a more customized datapath to memory. These changes improve performance, reduce resource overheads, and offer increased convenience to the application developer through an extensible library of data-structure-specific application-level memory interfaces.

For example, streaming a block of data from memory is a common operation. A

CoRAM++ application-level interface for streaming provides control threads with commands relevant to initiating and monitoring data streaming operations between DRAM and the hardware kernels. The hardware kernels use stream-specialized wire-level port interfaces that resemble a FIFO rather than an SRAM, in that the hardware kernels can only access the next item in the sequence.

An application-level interface used to interact with other data structures would be similarly specialized. For example, the application-level interface for a multi-dimensional array would provide commands for array traversal operations along various dimensions and support selecting between multiple in-memory data layouts at run time. The corresponding wire-level port interface could provide either FIFO-style ports or SRAM-style ports.

Below this abstraction, CoRAM++ allows data-structure-specific application-level interfaces to generate a streamlined soft-logic datapath to memory that can selectively instantiate components in order to improve performance. These performance benefits can be especially noticeable for irregular pointer based structures. For example, the application-level interface for a linked list could attach a hardware linked list engine directly to a DRAM interface to accelerate linked list operations by minimizing the latency of pointer chasing operations.

1.4 Thesis Contributions

CoRAM++ encompasses a data-structure-specific approach to application development, an FPGA application development environment supporting this approach, and an extensible library of data structures. This thesis demonstrates that CoRAM++ supports

conveniently supports a variety of data-structure-specific application-level interfaces without introducing too much overhead in terms of run-time performance or resource utilization, making the following contributions:

1. The CoRAM++ approach to FPGA application development, which is designed around a decoupled computing model that supports an extensible library of data-structure-specific application-level memory interfaces.
2. An application development environment supporting this computing model, which layers this library of data-structure-specific memory interfaces on top of a low-level system-level interface, and allows library components to instantiate custom datapaths to memory (including custom modules directly connected to memory interfaces) in order to deliver better performance when supporting irregular, pointer-chasing data accesses.
3. Initial work towards a library of data-structure-specific memory interfaces supporting data access patterns that are important in FPGA computing.
4. A demonstration that the CoRAM++ programming environment makes application development easier without undue overhead in terms of run-time performance or resource utilization.

1.5 Thesis Outline

The remainder of this thesis presents the details of CoRAM++, covering both its inner construction and external APIs. Chapter 2 provides background information on the state of the art in FPGA application development. Chapter 3 discusses the CoRAM++ programming environment in detail, and Chapter 4 presents the current library of application-level

interfaces supported by CoRAM++. Chapter 5 discusses the implementation details of the CoRAM++ environment, and Chapter 6 details the implementation of the CoRAM++ interface library. Chapter 7 details benchmark applications that have been built using CoRAM++. Finally, Chapter 8 concludes and proposes future extensions to CoRAM++.

Chapter 2

Background and Related Work

This chapter summarizes prior work on FPGA computing, programming environments, other tools support the development of FPGA computing applications.

2.1 Computing with FPGAs

In addition to their common use as prototyping devices, FPGAs have long been studied as general purpose computation devices. Numerous programming models for computation with reconfigurable devices have been explored, including using reconfigurable logic to implement custom instructions within processors [19], as coprocessors paired with a microprocessor [45], shared between several processors within a multi-core chip [80], connected to a multiprocessor system through the memory bus [21], and as standalone devices connected to Ethernet networks [25]. FPGAs are especially good at power-efficient computing [26, 30, 40, 78].

In spite of the demonstrated abilities of FPGAs as computation devices, very few FPGAs devices are employed for general purpose computation [9]. The reason that this is the

case is that FPGAs are difficult to use. FPGAs applications are most commonly designed using Register Transfer Languages (RTLs) such as Verilog and VHDL, which require the application designer to manually manage fine grained parallelism at a very low level of abstraction. It is particularly cumbersome to specify sequential actions through this programming model, which requires manually defining states and state transitions, and then manually encoding the states and transitions into register actions. The RTL programming model increases the burden required interact with DRAM and other communication interfaces (such as PCI, Ethernet, USB, and serial ports), as these interactions generally require managing sequential operations.

In addition, DRAM interfaces and other communication interfaces also vary widely between FPGA devices, meaning that the effort that went into making an application work well with the DRAM interface on one FPGA chip must be repeated when transitioning the application to another FPGA. The requirement of re-creating the logic that connects to DRAM interfaces not only applies when moving to an FPGA from a different manufacturer, but also applies to newer devices from the same manufacturer. The remainder of this section will discuss various strategies that have been used to raise the level of abstraction in FPGA application design, including programming environments that layer abstractions on top of FPGAs to simplify application development and provide portability.

2.2 Soft-logic and Hard-logic Processor Cores on FPGAs

FPGA development toolkits provide soft-logic processor cores [13, 88], and some FPGA chips include hard-logic processors cores that run much faster than soft-logic cores and can be used without consuming reconfigurable fabric resources [14, 86]. FPGA devel-

opment tools include a software development kit to allow application developers to write software to run on these processor cores, and include a bus interconnects and software toolkits that allow communication between processor cores and custom logic mapped to the FPGA's reconfigurable fabric. Soft processors offer limited performance due to low clock speeds, and can only connect to user-defined logic modules through buses. These buses increase the latency between DRAM and user-defined modules, mitigating the ease-of-use advantages that they provide, and can make it difficult to utilize the full bandwidth that DRAM controllers provide.

Hard processor cores do not necessarily fare any better in terms of achievable DRAM bandwidth, even though they can run at much higher clock speeds. FPGA chips such as the Xilinx Zynq [86] and Altera Arria SoC [14] contain two hard-logic ARM processor cores, hard-logic caches, a hard-logic memory controller, and hard-logic data distribution network, and connect the reconfigurable fabric to the data distribution network through multiple ports. The hard-logic components run at a higher clock rate than they would achieve if implemented in soft logic, but while they support high speed DDR-3 interfaces, Xilinx limits the physical width of the Zynq's hard logic DRAM interfaces to 32 bits. This is half the typical DDR3 chip-to-chip interface width, limiting the ARM cores and reconfigurable fabric to half the bandwidth that these DRAM interfaces would typically provide, since Xilinx runs the interfaces at the typical DDR-3 clock speeds. Some of Altera's Arria SOC chips also limit the DRAM interface to a 32 bit width, but run the interface at twice the clock speed of their Xilinx counterparts, and are less bandwidth constrained.

Both Xilinx and Altera FPGAs with built-in hard processor cores can include addi-

tional DRAM interfaces besides the DRAM interfaces attached to the hard processor cores, which can provide higher bandwidth to custom logic in the reconfigurable fabric, However, neither FPGA manufacturer includes functionality to help the application interact with these DRAM controllers (beyond their standard DMA engines), forcing application developers to interact with the faster DRAM controllers the same way that they would if the ARM cores were not present.

Intel has released Stellarton [49] chip, which included an Atom CPU and FPGA fabric in the same package. This chip connected the FPGA fabric to the remainder of the system through a slow first generation PCI-Express connection – it did not have any access to DRAM. More recently, Intel has announced new tightly coupled CPU-FPGA systems [35], which use a cache coherent connection (over a bus called "QPI") allowing the FPGA to access DRAM. These systems are just becoming more widely available, and provide a coherent cache on the FPGA side of the system, but do not provide higher level data management mechanisms.

2.3 Higher Level Hardware Description Languages

The two dominant hardware description languages used for FPGA design—Verilog and VHDL—require designing applications at a very low level, describing the interactions of circuit elements per clock cycle, and manually coordinating between them. While they do support some modularity and template creation, support for these features varies widely across RTL compilation software.

Several higher level languages for hardware development have been created. These languages are commonly based on functional programming languages, which are compiled

to HDL code that can then be processed using the typical hardware design tools. Examples of these languages are Bluespec [66], Chisel [20], and MyHDL [65]. These tools have been used for several large projects, up to and including processors [28, 36], and these tools are increasing in popularity over time. Several prior FPGA computing programming environments, including LEAP [38] and the CoRAM architecture [31], use Bluespec to assist with their implementation.

2.4 High-Level Synthesis Tools

High level synthesis is the process of generating hardware designs suitable for implementation on FPGAs from even higher level software source code. Numerous high level synthesis tools have been developed, including Xilinx Vivado High Level Synthesis (formerly AutoPilot and AutoESL) [85], Altera's C2H compiler [12], the Riverside Optimizing Compiler for Configurable Computing (ROCCC) [79], Impulse-C [48], Handel-C [59], and Catapult-C [58].

These tools are able to generate hardware pipelines from software source code, but have some limitations: High level synthesis tools generally do not have the ability to directly manage DRAM controllers (although Altera's tool does allow this and supports pipelined data transfers), and generally rely on the application designer to annotate the software source code in order to capture parallelism and data reuse. These tools also tend to be somewhat limited in the types of source code that they accept, although they are markedly improving over time. Ultimately, the capabilities of these tools are tied to their ability to extract parallelism and pipeline parallelism from sequential code.

There is recent work that utilizes AutoPilot with a variety of polyhedral optimizations

in order to capture parallelism [98], and our own work [81] has shown that the CoRAM abstraction, in concert with simple loop nest optimizations, can produce full implementations of high performance computations that interact with DRAM controllers.

LegUp [24] represents another class of high level synthesis tools. Source code is compiled to a hybrid architecture containing a MIPS soft processor, similar to the soft processors discussed above. The application is then profiled in order to detect hot areas of code, which are converted into custom accelerators.

Restricting the types of applications that a tool will support to a specific domain is one way to make the problem of automated system generation from a high level description more tractable. Domain specific tools have been created to build network processors [54], compile GPGPU applications [70], accelerate MATLAB code [43], implement linear processing transforms [60], and even generate custom FPGA devices [44].

2.5 Bus infrastructures and Networks-on-Chip

FPGA computing applications generally include infrastructure for data distribution within the chip. This infrastructure often comes in the form of buses such as AXI [17], but some infrastructure layers can produce point-to-point connections. Adding an infrastructure layer does introduce some overhead, but as some logic is necessary to implement the functionality provided by computing infrastructures, the additional overhead introduced by a well-written infrastructure layer is minimal.

Both Xilinx and Altera provide "system builder" tools that allow users to combine components using a GUI tool [16, 90]. These tools support load-store and streaming interfaces. and allow application developers to manually incorporate descriptor-based DMA

engines [92]. These tools can even support the hard cores and interconnects present on the chip. These tools manage all of the components connecting to a bus, multiplexing multiple interfaces into a single connection using a global address space, and arbitrating between multiple components that connect to DRAM and other communication interfaces. These tools are vendor specific, and it can be even difficult to transition designs between different versions of the tool, which don't always provide an automatic upgrade path for IP modules. It can also become cumbersome to use these point-and-click tools to create large applications.

As FPGAs grow larger and the number of components in use grows, there has been increased interest, from both the commercial and research communities, in replacing shared buses with Networks-on-Chip (NoCs) [46, 71], and even in hardening the NoC [6].

2.6 FPGA Computing Application Programming Environments

FPGA computing application programming environments attempt to simplify FPGA application development by automating the process of creating data paths and providing services for moving data across interfaces and within the FPGA. Existing programming environments have primarily focused on simple, regular data access patterns such as streaming and block copy. They tend to use a Network-on-Chip (NoC) for data distribution, mapping applications to a "dance hall" style architecture. A "dance hall" architecture instantiates application components on one side of the NoC and DRAM controllers and communication interfaces on the other, requiring a round trip through the NoC for every data access.

Redsharc[52] provides stream and block APIs to kernels using custom hardware or

on an embedded processor, and also decomposes applications into control and compute, managing global control through software running on an embedded processor.

Maxeler MaxCompiler [57] supports streaming and array operations, and like CoRAM and CoRAM++, decomposes applications into kernels that compute and manager modules that orchestrate computation and I/O. MaxCompiler includes a Java-like language that can be used to assemble kernel pipelines, and supports address generation modules for multi-dimensional arrays.

RCMW[50] is a virtualized FPGA development environment that supports APIs for stream and burst transfers, mapping applications to one or more physical FPGAs. RCMW allows application developers to adjust interface properties and arbitration schemes, but does not support custom logic near DRAM controllers and communication interfaces.

LEAP[1] [38] is an operating system for FPGAs that supports many features, incuding data transport, barriers, and file I/O including standard input and output. LEAP applications consist of modules, which can be created out of hardware or software, that communicate through latency-insensitive communication channels. LEAP supports a memory hierarchy with a "scratchpad" interface, which consistof a NoC-attached hierarchy of caches[7], that can coherently share data between multiple hardware kernels [94]. LEAP can map application components onto multiple FPGA chips, embedded processors, and computers [39], and supports a prefetcher to improve the performance of strided accesses [93]. This prefetcher is transparent to the application, and cannot be tuned for specific data access patterns. The prefetcher is instantiated near application components in the NoC, rather than near the DRAM controllers. LEAP's only model for accessing mem-

[1]LEAP stands for both Logic-based Environment for Application Programming and Latency-insensitive Environment for Application Programming

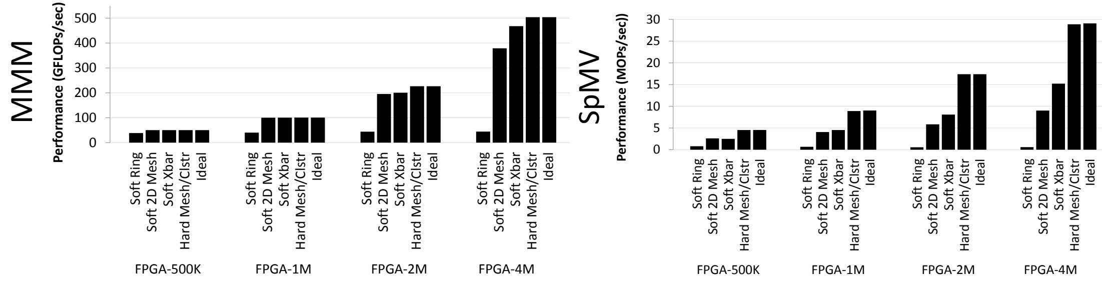
ory is a latency-insensitive load-store interface, although there is ongoing work on using LEAP as a system layer for CoRAM++, and modifying LEAP to support the CoRAM++ concept of accelerators that are directly connected to a memory interface.

Catapult [74] is a recent effort to bring FPGA computing into the mainstream, and is currently part of a large scale effort by Microsoft to move a research effort into production. It provides a standard "shell" interface on the FPGA device that handles off chip communication, and provides compute modules with a streaming interface. Catapult will greatly change the face of FPGA computing, as Microsoft's first goal with it is to double the performance of Bing search, which, if successful, will commercially use FPGAs on a scale at which they have not been used before. The current Catapult shell attaches FPGA DRAM interfaces directly to the application, and does not appear to abstract the raw DRAM interface in any way.

Most similar to CoRAM++ in motivation is APMC [47], which goes beyond simple data access patterns by including a specialized descriptor-based DMA engine supporting load-store, streaming, array, linked list, and tree based data transfers. APMC allows application developers to write software code to manage data transfers, which runs on a Xilinx microblaze processor. APMC's descriptor-based DMA engine includes a scratchpad that can be used as a traversal cache [33] to store the results of a data structure traversal for later reuse.

2.7 The CoRAM Architecture for FPGA Computing

The CoRAM architecture proposes new FPGAs with built in data management features (implemented in the hard logic that makes up a CoRAM FPGA), along with a pro-



(a) Matrix-matrix multiply results (b) Sparse matrix-vector multiply results

Figure 2.1: Simulation results using the original CoRAM architecture [32].

gramming model that decomposes applications into kernels that perform computation and control threads that manage control and communication. CoRAM defines a fixed set of application-level interfaces that are meant to be used as building blocks for all applications. These application-level interfaces include block copy memory APIs to move data between DRAM interfaces and kernels, and queue-based and register-based messaging APIs for communications between control threads and kernels. The CoRAM architecture supports "personalities" for layering application-level interfaces on top of its own, but does not allow CoRAM personalities to provide the most convenient interfaces to applications, as will be discussed in Section 2.7.3.

2.7.1 Mapping the CoRAM Architecture to Soft-Logic FPGAs

The CoRAM architecture has been realized in soft logic [32], and can perform well when supporting computations that perform regular data accesses. The chart on the left side of Figure 2.1 shows the results of a simulation study on a single precision floating point matrix-matrix multiplication (MMM) on the CoRAM architecture. The results are grouped by simulated FPGA size, ranging from 500k to 4 million logic elements. Each FPGA size includes performance results simulating a soft implementation of CoRAM using a ring, mesh or crossbar NoC, along with a hard implementation using a hard logic

mesh NoC and CoRAM cluster, and an "Ideal" result which is the peak theoretical throughput of the processing elements. In these simulation results, the performance of the soft-logic realizations of the application are almost identical to the hard-logic realizations for the two smaller FPGA sizes (which are still medium-to-large FPGAs when compared to commercially available chips in 2015), and the soft-logic realizations are close in performance to the hard-logic realizations when using the crossbar NoC in the two larger simulated FPGAs.

While the performance of the MMM computation (with regular memory accesses) is good when using a CoRAM implementation that is mapped to soft logic, the performance of irregular computations (which require data dependent memory accesses) is not. The chart on the right side of Figure 2.1 shows how the soft-logic implementation of the CoRAM architecture suffers when computing a Sparse Matrix-Vector Multiplication (SpMV). The figure shows the results of a SpMV using single precision floating point data stored in the common compressed sparse row (CSR) format [3], which omits zero-valued data in a sparse matrix to skip unneeded computations and data transfers. Omitting zero-valued data greatly reduces the number of computations to perform, but requires irregular, data dependent accesses in order to process the non-zero data. The figure shows that all of the soft-logic realizations of the application are much slower than the hard-logic ones when performing the SpMV calculation. The soft-logic realization of the CoRAM architecture can include a cache between the DRAM interfaces and NoC, but the lower clock speeds of the soft-logic realization of CoRAM does not allow the cache to overcome the latency of the NoC round trip for every access. One of goals of CoRAM++ is to resolve this type of problem by allowing CoRAM++ data-structure-specific application-level memory

interfaces to attach data-structure specific accelerators directly to the memory interface in order to make irregular, data dependent memory accesses faster. CoRAM++ also allows data-structure-specific application-level memory interfaces to run pointer-chasing code on hard-logic processor of tightly-coupled CPU-FGPA systems (such as the Xilinx Zynq or Intel HARP), using the higher clock speeds and hard-logic caches available on these systems to make pointer-chasing data accesses faster.

ShrinkWrap [29] is an add-on that tunes the NoC used by the soft-logic implementation of CoRAM for the application. ShrinkWrap allows the application to specify how much bandwidth various components need, and uses this information to configure the NoC topology and data path width for this bandwidth in order to reduce NoC resource requirements. Shrinkwrap can also create separate NoCs for components that only read data or only write data, creating a pair of trees that transport data in each direction, are tuned to the aggregated NoC bandwidth needed by application components while reducing the width of the data path going to each component, further reducing resource requirements. The evaluation of ShrinkWrap showed that when an application can benefit from a reduced-bandwidth NoC, ShrinkWrap can reduce resource requirements by an order of magnitude while increasing the maximum clock speed of the application.

2.7.2 C-to-CoRAM: CoRAM as a High Level Synthesis Target

The C-to-CoRAM compiler [81] is an exploration of the original CoRAM architecture as a high-level synthesis target, in order to understand the ramifications of the CoRAM architecture's decoupled computing model on the high-level synthesis process. Most of the high-level synthesis tools discussed above do not support interactions with DRAM, and the goal of C-to-CoRAM was to see if the techniques that they use to generate hardware from

software code could be combined with the CoRAM architecture's data transfer abstraction to generate complete applications.

Another important distinction between C-to-CoRAM and the high-level synthesis tools discussed above is that CoRAM++ is annotation-free. Most high-level synthesis tools require that the user specify which parts of the application to pipeline and parallelize, but the C-to-CoRAM compiler does not. This ability is due to the C-to-CoRAM compiler's focus on loops—by restricting the code that the compiler processes to tight loops. it can find code that be pipelined and parallelized by tracing through data dependencies.

Figure 2.2 shows the workflow used by the C-to-CoRAM compiler. The C-to-CoRAM compiler takes a loop nest program, partitions the program into compute and control portions, and the uses the ROCCC [79] high-level synthesis tool together with the soft-logic realization of the original CoRAM architecture to generate an FPGA programming file. The C-to-CoRAM compiler uses well known auto-parallelization and data reuse detection techniques in order to create applications that can perform well on FPGAs. The C-to-CoRAM compiler can automatically interleave independent computations through deeply pipelined kernels, reorganizes data transfers for efficient implementation, and instantiates stream buffers to take advantage of data reuse that it finds.

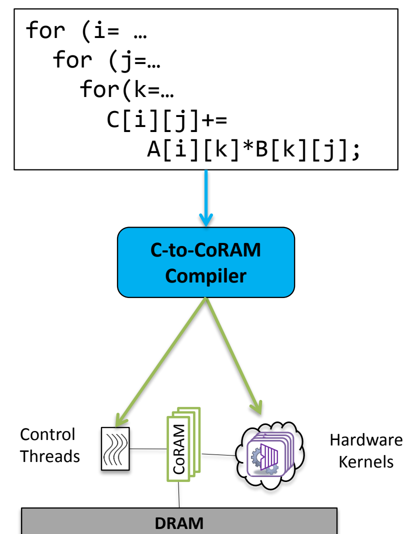


Figure 2.2: C-to-CoRAM compilation workflow.

While reorganizing data transfers, the C-to-CoRAM compiler converts them into a data stream that is delivered to hardware kernels, and can even generated stride-permutation

stream buffers to avoid strided accesses that inefficiently use the DRAM row buffer, by performing them on chip using local buffers that can retrieve any row with one cycle latency. These stream buffers instantiated by the C-to-CoRAM compiler are tuned for the exact data access pattern that the application needs. The compiler allows users to easily scale the generated hardware to fit larger or smaller FPGA devices, and target devices from multiple FPGA vendors.

The C-to-CoRAM compiler was tested using FPGA hardware from Xilinx and Altera on floating point matrix-matrix multiplication, 2d direct convolution, and *k*-nearest neighbor workloads, using one or two DRAM controllers. Application performance on FPGA hardware ranges from $4\times$ slower to $2\times$ faster than previously published work on the same hardware. It can use the ShrinkWrap extensions to the original CoRAM architecture to reduce NoC requirements when not all of the DRAM bandwidth is needed.

The main takeaway from the C-to-CoRAM project is that it is possible to use the original CoRAM architecture as a high-level synthesis target. As with many high level synthesis tools, C-to-CoRAM can only provide good performance on code from which parallelism and data reuse can be automatically extracted. C-to-CoRAM extracted parallelism and data reuse by recognizing particular data access patterns and instantiating pipelined and parallel computation kernels and parameterized stream buffers to take advantage of these data access patterns. The C-to-CoRAM compiler could use state-of-the-art polyhedral analysis techniques to find and parallelize more code, but by doing so it would likely lose the ability to perform annotation-free optimization, as most polyhedral analysis tools require annotation to restrict their search space.

2.7.3 Limitations of the CoRAM Architecture and its API

The CoRAM++ abstraction inherits its high level programming model from the original CoRAM architecture, which is the decomposition of applications into Processing Elements that perform computation and Control Threads that manage global control and data transfers. This decomposition allows a separation of concerns between compute and control, and allows Control Threads to be expressed through a software design language rather than a hardware design language. Since Control Threads are meant to preside over data transfers that take many cycles and are limited by the DRAM controller, they would have little to gain from the performance increase provided by the cycle-level control of hardware design languages.

This programing model causes performance issues in soft logic when an application performs data dependent accesses – for example with the sparse matrix-vector multiplication whose simulation results are discussed above in Section 2.7. The simulation results show that when targeting the higher clock speeds of hardware the CoRAM architecture does much better than when targeting soft logic, in spite of the fact that the soft-logic realization of the CoRAM architecture includes a cache attached to the DRAM controllers. The reason for this issue is the latency of DRAM requests, which much start from a control thread (which has been compiled to an FSM), go across the NoC, through the cache (when included), and to the DRAM interface. The data-structure-specific approach of CoRAM++ addresses this issue by allowing data-structure-specific accelerators that are attached directly to the DRAM interface. Chapter 5.1 will provide more detail about data-structure-specific accelerators.

In addition to providing better performance, CoRAM++ also provides more conve-

nience to the application developer through simple data-structure-specific application-level interfaces with “fire-and-forget” semantics—that is application developers can issue commands using these interfaces, and rely on the CoRAM++ programming environment to execute the commands without blocking the application, even if the command takes a long time to run. The original CoRAM architecture does not allow simple “fire-and-forget” style data structure code, as will be explained below.

As a hardware architecture, the original CoRAM architecture contains built-in support for a limited set of application-level interfaces. The original CoRAM architecture defines a block copy application-level interface that moved data between local SRAM blocks (called “CoRAM” blocks) and a global address space that contains DRAM controllers and communication interfaces The original CoRAM architecture also supports bidirectional communication channels between a single Control Thread and a single Processing Element in the form of message queues and shared registers. These application-level interfaces are intended to serve as building blocks for all communication patterns.

The original CoRAM architecture defines “personalities” as a way for library code to supplement its built-in application-level interfaces. CoRAM personalities can encapsulate CoRAM blocks and communication channels, allowing CoRAM personalities to build new application-level interfaces on top of the built-in SRAM interface, such as a FIFO-style interface allowing access to a sequence of data items. CoRAM personalities can also provide library code to application control threads, exposing function calls that allow application control threads to issue commands specific to the CoRAM personality. However, these function calls run within the application control thread, which can cause issues when a command triggers a long-running action.

Consider a CoRAM personality for a stream. A stream personality should expose a FIFO-style interface to hardware kernels, and allow control threads to issue commands that initiate and monitor stream operations. Within the CoRAM personality, stream operations are mapped onto block copy operations by using the SRAM as a circular buffer. The circular buffer will internally maintain head and tail pointers into the buffer, and will not allow writes into the buffer when the buffer is full, or reads when the buffer is empty. The need for these blocking flow control mechanisms prevents CoRAM personalities from providing the most convenient possible application-level interfaces to applications. Section 3.2 discusses the additional code that CoRAM control threads must include when using personalities for complex, long running operations.

Chapter 3

The CoRAM++ Programming Environment for FPGA Computing

3.1 Arguments for Specializing in a Soft-Logic Context

Like most of the programming environments for FPGA computing discussed in Section 2, the original CoRAM architecture was designed around support for a single data access pattern—block copy. The original CoRAM's native application-level interface restricts hardware kernels to obtaining data from wire-level ports that connect to SRAM blocks, and restricts control threads to issuing commands that transfer data between DRAM and small SRAM blocks.

While it is possible to map all data access patterns onto a block copy paradigm, data-structure-specific application-level interfaces can be more convenient for the application developer. For example, in the original CoRAM a streaming data transfer can be implemented through repeated block copy commands issued by a control thread. However,

the control thread could be simplified if it could issue simple "fire-and-forget" streaming commands to initiate and monitor streaming operations. These simple commands also allow the control thread to avoid the burden of handling subtle issues, such as flow-control related deadlocks.

The hardware kernel could also benefit from specialized interfaces for streaming. Rather than explicitly managing an SRAM as a circular buffer, the hardware kernel could utilize an application-level interface providing FIFO-style ports.

The data-structure-specific approach of CoRAM++ can also improve performance. The original CoRAM's simple, generic application-level interface is intended to serve as the universal primitive supporting all memory access patterns, and would perform well in a hardened implementation within a future FPGA. But a soft-logic implementation of CoRAM on today's commodity FPGA suffers in performance and resource overheads due to the inherent inefficiencies of soft logic. A data-structure-specific approach can recoup some of these losses in addition to providing a more convenient application-level interface.

Consider a hardware kernel operating on data stored in a linked list. Traversing the linked list requires pointer chasing operations to follow the chain of linked list nodes, and may also require pointer chasing to retrieve data payloads. While pointer chasing operations are feasible under the original CoRAM application-level interface, each pointer chasing operation must be performed by the control thread using explicit DRAM operations with very long latencies.

A command set that directly operates on linked lists not only simplifies the control thread, but also permits performance optimization by the underlying implementation. A linked list application-level interface can include a linked-list-specific engine that is be

connected directly to a DRAM interface. As will be explained in Section 3.4 below, this module can make linked list operations faster by minimizing latency when following the linked list pointers, and batching data transfers across the NoC. This linked list application-level interface might provide better even performance than could be achieved by a hardened implementation of CoRAM, which would not allow custom soft-logic modules to be connected directly to a DRAM interface, and consequently would incur many round trips across the NoC when traversing a linked list.

Data-structure-specific application-level memory interfaces can also provide interesting opportunities for hardware-software codesign. For example, Xilinx's Zynq and Intel's HARP are tightly coupled CPU systems—the FPGA fabric access the same DRAM as a CPU core, and can coherently share memory. When running an application with with pointer-chasing memory accesses on a these systems, it might be advantageous to run pointer-chasing operations on a CPU core, which can stream data to the FPGA fabric through either a dedicated hardware port or a DRAM-based queue. The application would benefit from the higher clock speeds and hard-logic cache of the CPU core in this scenario, as well as the ease-of-use of writing pointer-chasing code as software.

3.2 Complexities in Creating Extensible Data-Structure-Specific Application-Level Interfaces

Designing a programming environment to conveniently support extensible application-level interfaces (data-structure-specific or not) is not trivial. Library code must map all data transfers onto primitives that are directly supported by the underlying programming environment, and would ideally be able to completely mask their implementation in order

to support the most convenient interface possible.

For example, the original CoRAM architecture supports a supports a block copy data transfer primitive. Block copy is universal, but even the most simple data access pattern that is not block copy—streaming contiguous blocks of data to or from DRAM—cannot be built on top of a block copy data transfer primitive without some care. This issue arises because the underlying buffers used for copying blocks of data are finite in size, and flow control mechanisms must be used to ensure that data is not copied into buffers unless the buffers contain sufficient room for the data.

Streaming data transfers are commonly built on top of buffers using a circular buffer, using one pointer into the buffer as the location for writing data and another for reading it. The streaming data transfer library would keep track of both buffers, and prevent data from being written to the buffer unless the buffer contains space for the data. A streaming data transfer component should hide this implementation detail from the application, presenting the application's hardware kernel with FIFO-style ports, and providing control thread commands that initiate and monitor streaming data transfers.

Listing 1 depicts the desired control thread code for a streaming application written for the original CoRAM, using a CoRAM "Personality" to manage data transfers. The corresponding hardware kernel code is identical to the example in Chapter 3, and is omitted in the listing because the hardware kernel code is not the source of the issue at hand. Lines 1-7 and 13-14 are boilerplate—including headers, declaring a function and local variables, creating a loop. Lines 8-12 contain the useful parts of the control thread. The control thread reads a run-time parameter from the host computer, indicating the number of data items to transfer. The control thread then initiates a streaming transfer that writes data

```
1 #include "StreamPersonality.h"
2 #include "CommPersonality.h"
3 void streamer(int IN_ADDR, int OUT_ADDR) {
4   Personality s_in=getStreamPersonality(1);
5   Personality s_out=getStreamPersonality(2);
6   Personality comm=getCommPersonality(3);
7   while (true) {
8     unsigned count=read_comm(comm)*sizeof(unsigned);
9     write_stream(s_in,&IN_ADDR,&count);
10    read_stream(s_out,&OUT_ADDR,&count);
11    wait_stream(s_out);
12    write_comm(comm,0);
13  }
14 }
```

Listing 1: Streaming application code (which deadlocks using CoRAM stream personalities)

from DRAM to the “input” stream, and another that reads data from the “output” stream to DRAM. Finally, the control thread waits until data has been written to the “output” stream, and sends a signal to the host computer.

This control thread will not work as intended, because CoRAM personalities perform data transfers in the context of the application control thread. This prevents CoRAM personalities from performing long-running complex data transfers on behalf of the application without blocking the application control thread. This causes problems when a single control thread manages two stream personalities, because of the need for flow control when using a circular buffer to perform a streaming data transfer.

When the control thread in Listing 1 reaches line 9, it will invoke the stream personality code for that writes data from DRAM to the stream. This code will write a block of data to the circular buffer, and wait until the buffer has more space—that is until the kernel removes the data. At the beginning of the computation, the kernel will remove data, feed the

data into a streaming kernel, and eventually write output data to the output stream. As the kernel removes data from the stream, the control thread will continue issuing commands to copy data into the buffer. However, at some point, the output stream will fill up, because the output stream is also backed by a circular buffer. The only way for that space will open up in the output buffer is if the control thread issues commands to move data from the output stream to DRAM, which will not happen until the control thread reaches line 10. But the control thread cannot reach line 10 until it issues all of the commands necessary to write the entire input data set to the input stream on line 9, leading to a deadlock.

This deadlock is ultimately caused by the fact that CoRAM personalities manage data transfers in the context of application control threads—that is CoRAM personalities cannot manage data transfers independently of application code. There are two solutions available to CoRAM applications that resolve this issue: writing application code that manually invokes callback functions to manage multiple data transfers, or writing application code that manually creates a separate control thread for each stream. Each solution is discussed below.

Listing 2 illustrates a callback-based streaming application. The core of this application is an additional application loop that calls non-blocking functions that issue commands to each stream, and tracks the amount of data written to and read from the stream individually. While it is only four lines longer than the code in Listing 1, the extra loop makes it conceptually much more complicated.

Listing 3 presents the multi-threaded solution. This code creates two separate control threads, along with a communication channel between them. Each control thread manages a separate stream, and proceeds independently as it issues blocking data transfers. As with

```
1 // Control Thread code for a callback-based  memory
    personality-based streaming application
2 #include "StreamPersonality.h"
3 #include "CommPersonality.h"
4 void streamer_callback(int IN_ADDR, int OUT_ADDR, int
    SIZE) {
5   Personality s_in=getStreamPersonality(1);
6   Personality s_out=getStreamPersonality(2);
7   Personality comm=getCommPersonality(3);
8   while (true) {
9     unsigned count_r=read_comm(comm)*sizeof(unsigned);
10    unsigned count_w=count_r;
11    while (count_r>0) {
12     if (count_w>0)
13        write_stream_nonblocking(s_in,&IN_ADDR,&count_w);
14     read_stream_nonblocking(s_out,&OUT_ADDR,&count_r);
15    }
16    wait_stream(s_out);
17    write_comm(comm,0);
18 }
```

Listing 2: Streaming application code that uses application callbacks to avoid deadlock

the callback-based solution above, this code is more complex than the desired code in Listing 1. CoRAM++ allows library code to instantiate control threads that are independent from the application components. This feature allows “fire-and-forget” style commands exposing data-structure-specific application-level interfaces that are as convenient as possible.

3.3 CoRAM++ Scope and Underlying Assumptions

The CoRAM++ FPGA programming environment is primarily concerned with computations that run entirely within the reconfigurable FPGA fabric, and focuses on data movements between application components and DRAM. It would be possible to partition

```
1 // Control Thread code for threaded memory
    personality-based streaming application
2 #include "StreamPersonality.h"
3 #include "ThreadThreadPersonality.h"
4 #include "CommPersonality.h"
5 // This thread reads from the kernel and writes to DRAM
6 void streamer_read(int OUT_ADDR, int SIZE) {
7   Personality s_out=getStreamPersonality(2);
8   Personality comm=getCommPersonality(3);
9   Personality to_read=getThreadThreadPersonality(4);
10  while (true) {
11    unsigned count=read_comm(comm)*sizeof(unsigned);
12    write_to_other_thread(to_write,count);
13    read_stream(s_out,OUT_ADDR,count);
14    wait_stream(s_out);
15    write_comm(comm,0);
16  }
17 }
18 // This thread reads from DRAM and writes to the kernel
19 void streamer_read(int IN_ADDR,int SIZE) {
20  Personality s_in=getStreamPersonality(1);
21  Personality to_write=getThreadThreadPersonality(4);
22  while(true) {
23    unsigned count=read_from_other_thread(to_write);
24    write_stream(s_in,IN_ADDR,count);
25  }
26 }
```

Listing 3: Streaming application code for CoRAM using application threads to avoid deadlock

larger applications into components that run on an embedded processor (such as within the Xilinx Zynq [86]) or within a multi-socket CPU-FPGA system (such as Intel's new Xeon-FPGA hybrid system [35]), but CoRAM++ does not currently provide any tools to help an application developer decide how to partition the application between CPU cores and the FPGA fabric.

However, CoRAM++ does include some support these systems. Section 7.3.2 evaluates the possibility of accelerating pointer-chasing memory operations using the CPU cores of a tightly-coupled CPU-FPGA system. Since the CPU cores in such a system will run at higher clock speeds than the FPGA fabric and contain hard-logic caches, they should be able to perform pointer-chasing memory operations faster than the FPGA fabric. CoRAM++ also supports a host computer interface (see Section 4.2.2) that assists in moving data between a CPU-based system and FPGA DRAM. This host computer interface has a well-defined communication protocol, and uses the ARM cores on the Zynq FPGA to manage data transfers over an ethernet communication interface.

3.4 CoRAM++ Programming Environment Overview

Like the original CoRAM architecture, the CoRAM++ FPGA programming environment simplifies FPGA application development while supporting high performance data transfers. The CoRAM++ programming environment retains the decoupled programming model of the original CoRAM architecture. But CoRAM++ makes better use of the reconfigurable nature of soft-logic commodity FPGAs and generates a customized datapath to memory that is tuned for the data structures used by the application.

As discussed in Chapter 1.2, CoRAM++ applications are decomposed into hardware kernels that compute and control threads that control threads that sequence computation and communication. These components communicate using data-structure-specific application-level interfaces which are used to facilitate data transfers in an FPGA-agnostic fashion. An application developer designing an application for CoRAM++ performs the following steps:

1. Designs the hardware kernels.
2. Selects data-structure-specific application-level interfaces.
3. Writes control threads.
4. Runs the CoRAM++ compiler.
5. Invokes FPGA-vendor-specific tools.

CoRAM++ makes no restrictions on how hardware kernels are created—they can be created using any hardware design methodology, including IP core generators and the HLS generation tools discussed in Chapter 2. The data-structure-specific application-level interfaces are selected from an extensible library, and can be customized if necessary. The underlying implementation modules that support application-level interfaces are called "agents," because they act independently of application components to manage data transfers. Section 4 describes the data structures that are currently supported by this library.

CoRAM++ Control threads are written using a broad subset of the C language, which supports the common control flow constructs, and will be described in more detail in Section 5, which also discusses the CoRAM++ compilation process. CoRAM++ currently supports FPGAs from both Xilinx and Altera, and includes abstracting wrappers that provide a uniform interface to block memories and floating point cores.

3.5 Example CoRAM++ Application

This section presents a streaming CoRAM++ application that calculates a double precision Discrete Fourier Transform (DFT). A streaming application is convenient for elaborating CoRAM++ application components, but Chapter 7 demonstrates the CoRAM++ abstraction is not limited to streaming applications.

Since DFTs are common, an application developer would create a DFT core with an IP core generator (such as Spiral [61]) rather than by hand. When creating the application without using the CoRAM++ abstraction, the application developer would complete the application by providing:

1. An FPGA-specific module that instantiates DRAM controllers, clock generators, and off-chip communication interfaces.
2. Address generation logic.
3. A data distribution network.
4. Control logic that manages these components and the process of initializing FPGA DRAM.

The CoRAM++ abstraction significantly simplifies this process. When targeting the CoRAM++ abstraction, the application developer creates a kernel, which instantiates the DFT core and CoRAM++ components shown in Listing 4, and then writes the small amount of control thread code shown in Listing 5. The CoRAM++ compiler (discussed in Section 5.1) processes the application components and invokes vendor-specific tools to produce an FPGA programming file.

Figure 3.1 depicts the an application-level view of the streaming DFT application,

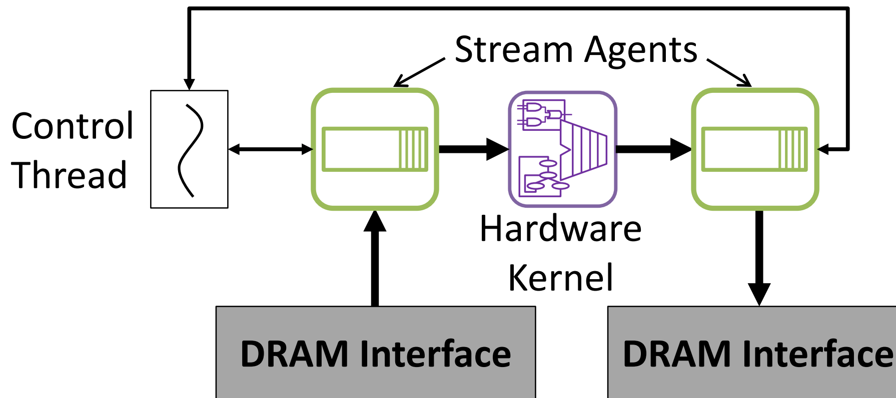


Figure 3.1: Application-level view of a streaming CoRAM++ DFT application. showning application components, DRAM interfaces, and two stream agents (which implement the stream application-level interfaces used by the application). A single control thread manages data transfers. The control thread directs one stream agent to deliver data from DRAM to the hardware kernel (containing the DFT core), and directs another stream agent to write output data to DRAM. The kernel interacts with FIFO-style wire-level ports exposed by the stream agents.

Listing 4 contains the Verilog source code for the hardware kernel. The application processes a pair of DFT inputs per cycle, each containing real and imaginary components. The hardware kernel instantiates the DFT core and several CoRAM++ agents. In general, CoRAM++ agents are parameterized by the name of the thread to which they connect and a unique identifier. The stream agents expose wire-level ports that are used within the hardware kernel, but the other two CoRAM++ agents do not expose ports as they do not interact with the Processing Element. The host computer agent declared in lines 15-17 allows the user to initialize FPGA DRAM and manage application execution from a host computer, and the application's control thread is instantiated in lines 18-21. Note that the hardware kernel is not concerned with ports for external connections—those that attach to

```
1 // dft.v: Processing Element code for the CoRAM++ DFT
2 module dft();
3  wire in_rdy, dft_rdy,out_rdy, dft_done;
4  wire[255:0] dft_in_data;
5  wire[255:0] dft_out_data;
6  // These Agents perform data transfers
7  ReadStream#(.ID(1) ,TID("dft"),.WIDTH(256))
8    inQ(.notEmpty(in_rdy),.first(dft_in),
       .deq(in_rdy&&dft_rdy);
9  WriteStream#(.ID(2) ,TID("dft"),.WIDTH(256))
10   outQ(.data(out_data),.enq(dft_done&&!out_rdy),
       .notFull(out_rdy));
11  // This IP core performs computation
12 DFTKernel#(.SIZE(64),.PRECISION(64))
13   kern(.in(dft_in),
       .out(dft_out),.issue(n_rdy&&dft_rdy),
14      .out_valid(dft_done),.stall(dft_done&&!out_rdy);
15 // The Host Computer Agent manages execution
16 // and interactions with the host computer
17 HostComm#(.ID(3), .TID("dft")) comm();
18 // This Agent instantiates the Control Thread
19 ControlThread#(.FNAME("dft"),
20   .PARAMS("IN_ADDR=0;OUT_ADDR=0x100000;SIZE=8192"))
21   appThread;
22 endmodule
```

Listing 4: CoRAM++ Streaming 64-input DFT Hardware Kernel.

a NoC and ultimately to the DRAM interfaces on the FPGA in use. These connections are created during the compilation processes, which finds and processes the needed interfaces in through a process to that used by Soft Connections [72].

The control thread in Listing 5 first imports the application-level-interfaces supported by the stream and host computer agents using standard #include directives, and attaches to them in lines 5-7. Line 9 shows how the application reads a user-specified run-time parameter through the host computer agent, and lines 10-12 show how the control thread

```
1 // dft.c: Control Thread code for the CoRAM++ DFT
2 #include "StreamAgent.h"
3 #include "CommAgent.h"
4 void dft(int IN_ADDR, int OUT_ADDR, int SIZE) {
5   Agent s_in=getStreamAgent(1);
6   Agent s_out=getStreamAgent(2);
7   Agent comm=getCommAgent(3);
8   while (true) {
9    unsigned count=read_comm(comm)*SIZE;
10   write_stream(s_in,IN_ADDR,count);
11   read_stream(s_out,OUT_ADDR,count);
12   while (!stream_status(s_out)) {}
13   write_comm(comm,0);
14  }
15 }
```

Listing 5: CoRAM++ Streaming 64-input DFT Control Thread.

initiates non-blocking data transfers and waits for their completion. Finally, line 13 shows how the application indicates that the computation is complete, and that all output data is in DRAM. In this example, the addresses of input and output data are hard coded, but these addresses could be run-time variables.

Chapter 4

Currently Supported Application-Level Interfaces

The CoRAM++ library of application-level interfaces is extensible and will grow over time. CoRAM++ currently supports data structures that are fundamental to computing [34] or critical to highly-valued application domains. In addition to data-structure-specific interfaces,, the CoRAM++ application-level interface library includes support for several "utility" interfaces. The underlying modules that implement these application-level interfaces are called CoRAM++ "agents." In general, the term "agent" will be used to describe the instantiation of an application-level interface or its implementation details.

4.1 Data-Structure-Specific Interfaces for Memory Access Patterns

The CoRAM++ data structure library currently supports application-level interfaces for streaming (sequential access to 1D arrays), multi-dimensional arrays, linked lists, and static graphs (which do not change during computation, and support graph traversal opti-

Interface Type	Name	Description
Hardware Kernel Ports	addr[]	Local SRAM address.
	read/write	0 if the current request is a read, or 1 if it is a write.
	din[]	Input data for write.
	dout[]	Output data for read. Output data is valid one cycle after the read request.
Control Thread Commands	write_nb(b,g_addr, l_addr, bytes)	Write *bytes* bytes of data from global address *g_addr* to local storage *b* starting at address *l_addr*. Returns a transaction tag that can be used to monitor the data transfer.
	read_nb(b,g_addr, l_addr, bytes)	Write *bytes* bytes of data local storage *b* starting at address *l_addr* to global address *g_addr*. Returns a transaction tag that can be used to monitor the data transfer.
	check_tag(tag)	Returns true if the transaction referred to by *tag* has completed

Table 4.1: Block copy application-level interface.

mization through preprocessing). CoRAM++ data-structure-specific application-level interfaces are divided into two parts: wire-level ports that are exposed to hardware kernels, and function calls that control threads use to send commands.

4.1.1 Block Copy Interface

The block copy interface allows applications to transfer small contiguous blocks of data between DRAM (or any communications device mapped into a global address space) and local storage the FPGA. This interface mimics the application-level interface of the original CoRAM architecture. However, unlike the original CoRAM architecture, this application-level interface does not specify that data be stored in actual SRAM blocks, but instead specifies an SRAM-like interface for hardware kernels that may support simultaneous access through multiple SRAM-style wire-level ports.

Interface Type	Name	Description
Hardware Kernel Ports	notEmpty	Indicates whether or not the read stream contains data.
	dout[]	The first data item in the read stream.
	deq	Dequeue the first item in the read stream.
Control Thread Commands	write_stream(s,a,b)	Write b bytes of data from global address a to write stream s.
	stream_status(s)	Indicates whether or not all operations on stream s have completed.

Table 4.2: Read stream application-level interface.

Table 4.1 describes the block copy application-level interface (showing a single hardware kernel port interface), including a single cycle of delay when reading data from local storage. The wire-level ports presented to hardware kernels mimic typical SRAM-style ports, and the control thread commands match those supported by the original CoRAM architecture.

4.1.2 Stream Interface

The stream interface supports streaming applications that transfer data sequential data to or from DRAM. There are two variants of this interface—one that streams data from DRAM (or a communication interface mapped into the global address space) to a hardware kernel, and one that streams data in the opposite direction. Streams can be configured (at compile time) to carry arbitrarily wide data.

The read stream interface (which reads data from DRAM) is presented by Table 4.2, and the write stream interface (which writes data to DRAM) is rpesented by in Table 4.3, The hardware kernel connects to FIFO-style ports, and the control thread can issue non-blocking commands that issue data transfers to or from the global address space, and determine the status of outstanding requests.

Interface Type	Name	Description
Hardware Kernel Ports	notFull	Indicates whether or not the write stream has room for a new data item.
	din[]	Input data to write to the stream.
	enq	Write the data *din* to the steam.
Control Thread Commands	read_stream(s,a,b)	Read *b* bytes of data from read stream *s* to global address *a*.
	stream_status(s)	Indicates whether or not all operations on stream *s* have completed.

Table 4.3: Write stream application-level interface

4.1.3 Multi-dimensional Array Interface

The multi-dimensional array interface is provides a FIFO-style interface to hardware kernels while providing control thread commands that are convenient for handling multidimensional array data. The application developer specifies the number of dimensions, dimension lengths, and size of sub-blocks at compile time. As with the stream interface described above, the multi-dimensional array interface supports two variants, one for reading data to memory, and one for writing data to memory If needed by an application, it would be easy to create a third variant of this interface that supports SRAM-style ports, allowing random access to blocks of array data.

Table 4.4 shows the read multi-dimensional array interface, and Table 4.5 shows the corresponding write multi-dimensional array interface. Like the stream interface, each instance of the multi-dimensional array interface must be configured either for reading from the global address space or writing to the global address space.

The multi-dimensional array interface can stream the entire array in any dimension, and the application can specify, at run time, whether the data is stored in row-major or tiled order, and whether to deliver data to the stream in row major or tiled order. The

Interface Type	Name	Description
Hardware Kernel Ports	notEmpty	Indicates whether or not the read stream contains data.
	dout[]	The first data item in the read stream.
	deq	Dequeue the first item in the read stream.
Control Thread Commands	write_stream (s,a, i[,j,k...])	Write an item at indices (*i*,*j*,*k*...) to stream *s* from the array starting at address *a*. Note that the number of dimensions is fixed at compile time.
	write_stream_row (s, a,i,d)	Write row *i* along dimension *d* from array starting at address *a* to stream *s*.
	write_stream_block (s,a,i)	Write block *i* from array starting at address *a* to stream *s*. Blocks are addressed in row-major order, and have the same dimensionality as the array.
	write_array (s,a,d,o, g,p)	Write entire array *a* by dimension *d* stored in orientation *o* to stream*s* with granularity *g* using permutation *p*. Orientation can be row-major or blocked, and specifies the storage order in memory, and granularity specifies whether to deliver rows or blocks of data to the stream.
	test_stream(s)	Test to see if all stream operations on stream *s* have completed

Table 4.4: Read multi-dimensional array application-level interface.

agent can also transfer individual data elements, or rows or columns (or a sequence of data elements along any dimension), or multidimensional tiles of data. Any combination of these traversals are valid, but performance will suffer if data is traversed in an order that requires striding through memory, which causes inefficient use of the DRAM row buffer. Section 7.2 illustrates the effects of re-ordering the data on the fly. The write multi-dimensional array interface supports the exact same functionality for writind data to memory.

Interface Type	Name	Description
Hardware Kernel Ports	notFull	Indicates whether or not the write stream has room for a new data item.
	din[]	Input data to write to the stream.
	enq	Write the data *din* to the steam.
Control Thread Commands	read_stream (s,a, i[,j,k...])	Read an item at indices (*i*,*j*,*k*...) from stream *s* to the array starting at address *a*.
	read_stream_row (s, a,i,d)	Read row *i* along dimension *d* to array starting at address *a* from stream *s*.
	read_stream_block (s,a,i)	Read block *i* along dimension *d* to array starting at address *a* from stream *s*.Blocks are addressed in row-major order, and have the same dimensionality as the array.
	read_array (s,a,d,o, g,p)	Read entire array *a* by dimension *d* stored in orientation *o* from stream*s* with granularity *g* using permutation *p*. Orientation can be row-major or blocked, and specifies the storage order in memory, and granularity specifies whether to deliver rows or blocks of data to the stream.
	test_stream(s)	Test to see if all stream operations on stream *s* have completed

Table 4.5: Write multi-dimensional array application-level interface.

Multi-dimensional arrays can be mapped to a linear memory address space in several ways. The most common data organization, such as row-major, is a linear organization in which data within rows of the array are contiguous in memory. This organization supports efficient traversal along one dimension, at the expense of inefficient strided traversals in other directions. The interface also supports a tiled data layout, in which data is laid out in a tiled Z-Morton [64] order. Figure 4.1 illustrates a tiled 2D data layout.

In addition to streaming data in application defined order, the application can configure the interface with one or more permutation engines, which can be selected at run time. Permutation engines can rearrange data in the stream in arbitrary ways, including trans-

Figure 4.1: A 2D tiled data layout. 2D blocks of data are stored contiguously, allowing efficient traversal along any dimension.

posing blocks or entire data arrays. Since the attached hardware kernel only interacts with a streaming interface, it is also possible for permutation engines to change the number of data items being transferred to the stream, removing or adding items as necessary.

CoRAM++ includes macros that help configure the Spiral permutation generator for use with the multi-dimensional array interface. These macros include one to transpose a block of data, one to convert a strip of blocks into a strip of rows, and a simple pass-through queue. These macros allow the Discrete Fourier Transform application (Section 7.2) to traverse a two dimensional array along both dimensions, transposing the data delivered to the hardware kernel, using only two permutation engines, or to select between operating on purely tiled data or converting row-major data to tiled data in transit during the first phase of computation, efficiently converting data into tiled format on the fly, and traversing data in tiled format for the second phase, returning the data (at a cost in terms of hardware resources and a minimal amount of run-time latency) to row major format at the end of computation.

4.1.4 Linked List Interface

Linked lists are the canonical example of a data structure requiring irregular accesses, because it generally not possible to predict the address stored in the "next" pointer. But linked lists have a well defined structure, and the performance of operations on linked lists are generally dependent on how quickly requests for linked list nodes can be made.

The CoRAM++ linked list interface manages linked lists with a compile-time defined format, and includes a hardware linked list engine that performs pointer-chasing operations. The linked list interface can be configured for different linked list node structures, including those with in-line linked list data and those whose nodes contain a pointer to linked list data. Applications can traverse a linked list, streaming linked list data to the application or from it, modifying the data in the linked list. The application can also merge a pair of sorted linked lists in place, and will support more functionality in the future. As will be discussed in Section 7.3, attaching this hardware linked list engine directly to a DRAM interface is the key to good performance, and Section 6.3 provides more detail into the hardware linked list engine's construction. The hardware linked list engine can either be attached to the part of the linked list interfvace that interacts with application components, or it can be attached directly to a DRAM interface to reduce the latency of pointer-chasing operations as much as possible. The two configurations support the same application-level interfaces, allowing the different configurations to be benchmarked with minimal application changes.

The linked list engine can also run on the processor cores of a hybrid CPU-FPGA system such as Xilinx's Zynq. Section 7.3.2 presents an exploration of performing pointer-chasing linked list operations on a CPU core and streaming data to the FPGA fabric, com-

Control Thread Command	Description
initiate_request (l,op, ll_address0, ll_address1)	Initiates a linked list operation on interface *l* *Op* is a linked list operation that currently includes *read,write*, and *merge*. *ll_address0* is the start address for reading and writing data, and *ll_address1* is the start address of the second linked list when merging data.
query_status (l)	Queries the status of the current linked list operation on interface *l*. The query result includes whether or not the agent has completed the current operation, and how much data is available to read or write if traversing the list.

Table 4.6: Linked list application-level interface.

paring the performance of the hardware linked list engine and software, and several mechanisms for streaming linked list data to the FPGA fabric.

Table 4.6 presents the control thread portion of the linked list application-level interface. The application-level interface for the attached hardware kernel and CPU-based acceelrator are the same as the FIFO-style application-level interfaces described above. When the hardware linked list engine is directly connected to a DRAM interface, this engine is mapped into the global address space. The application can either use this application-level interface or can directly use drive the memory-mapped interface using a scratchpad agent (discussed below in Section 4.2.1) in concert with a stream interface that reads or writes linked list data.

4.1.5 B-Tree Interface

B-Trees are balanced trees that contain more than one key per tree node. This distinction is depicted in Figure 4.2, and distinguishes B-Trees from typical binary search trees [34]. B-Trees are useful for key-value storage because the balanced structure makes the pointer chasing accesses that must be performed when performing tree operations pre-

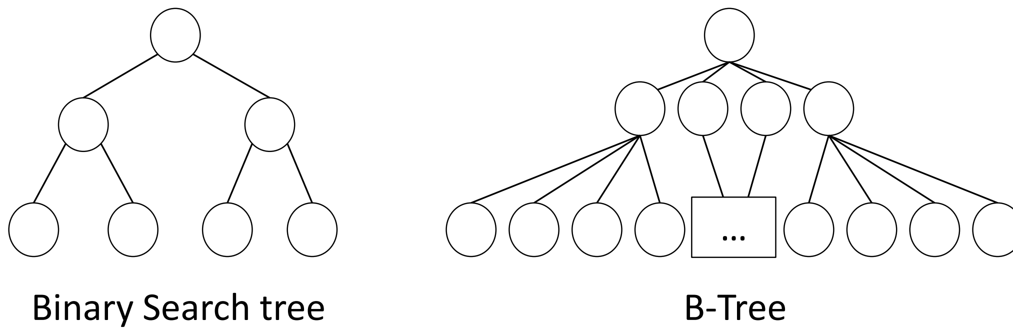


Figure 4.2: B-Tree structure as compared to typical binary search trees

dictable, and work well on systems with a multi-layer cache hierarchy as long as the size of the tree nodes is tuned to the cache hierarchy. There are many variants of B-tree-like structures, including B^+-Trees, which store all data values in leaf nodes, allowing a uniform tree node structure and ensuring that every access will take the same number of memory accesses,[1] and allowing backtracking during key deletion to be avoided. B-Trees are attractive for database operations because of their predictability and their ability to efficiently support range queries, which are less easily supported by hash-based indexes. B-Trees are subject to internal fragmentation within tree nodes when items are deleted, unless the tree is re-balanced, but the amount of internal fragmentation is subject to hard bounds.

The CoRAM++ B-Tree implements a variant of the B^+-Tree called a Preparatory Operation B^+-Tree [62], which preemptively splits tree nodes during insertion to limit backtracking. The CoRAM++ B-Tree does not re-balance the tree upon key deletion, due to recent theoretical results that show that re-balancing the tree upon key deletion is actually harmful to performance [76]. These theoretical results show that destroying and re-

[1] This is $log_b(k)$ where *b* is the number of keys in each tree node, and *k* is the number of keys in the tree.

Control Thread Command	Description
insert(key,value)	Inserts *key* and *value* into the tree, returning the previous value of *key* if *key* already existed or a token *invalid* if the key did not exist
find(key)	Finds the value associated *key* if *key* already existed or a token *invalid* if the key did not exist
delete(key)	Deletes *key*, returning its value or a token *invalid* if the key did not exist

Table 4.7: B-tree application-level interface.

creating the tree after a $\frac{1}{2}$ of the keys have been deleted is more effective than re-balancing it, and place hard limits on the amount of internal fragmentation and tree depth.

Table 4.7 shows the application-level interface for B-trees. Like the linked list interface discussed above, the b-tree interface includes a hardware b-tree engine that performs pointer-chasing operations and streams B-tree payload data to the application kernel (or a token indicating that the key in question was not part of the tree).

4.1.6 Gather-Scatter Work-List Interface

The gather-scatter work-list interface is designed for applications that break large computations into a sequence of computations on sub-sets of the data, which we will call a sub-task. Each sub-task operates on multiple data buffers, each of which may include hardware multiple ports. The gather-scatter interface fetches data from memory and places it into these buffers, triggers sub-task computation, and writing data back to memory. The application provides a list of gather operations, each of which requires following a pointer to transfer data to the local buffer (the size of each particular data transfer is encoded into the gather instruction). After sub-task execution is complete, the gather-scatter work-list interface will only write back buffers that have modified data, as a coarse-grained bandwidth-

Hardware Kernel Port	Description
raddr_*i*_*j*[]	Read address for buffer *i* port *j*
rdata_*i*_*j*[]	Read data for buffer *i* port *j*
ren_*i*	Read enable for buffer *i*
waddr_*i*[]	Write address for buffer *i*
wdata_*i*[]	Write data for buffer *i*
write_en_*i*	Write enable for buffer *i*
i_addr[]	Instruction address
i_data[]	Instruction data
start	Signal to hardware kernel to start computation
done	Signal from hardware kernel that computation is complete

Table 4.8: Hardware ports provided by the gather-scatter interface

saving measure. The gather-scatter work-list interface is double buffered, and overlaps computation with data transfers, and includes forwarding hardware that allows the gather-scatter interface to avoid data hazards by copying data between associated buffers between the execution of sub-tasks.

The gather-scatter interface is designed for applications in which can be broken up into sub-tasks prior to application execution, which allows the gather- and scatter-lists to be optimized for faster data transfers, coalescing small data transfers into larger ones in order to improve performance. The gather-scatter interface expects that one buffer is used to hold instructions determining each sub-task's execution, and defines an instruction format that specifies reads and writes to the local buffers. This pre-defined instruction format allows the gather-scatter interface's data processing script to reorder data within local buffers to improve data transfer coalescing opportunities, and to statically detect data hazards between adjacent sub-tasks and coordinate copying data between the buffers used by each sub-task.

Table 4.8 shows the hardware ports provided by the gather-scatter interface. This interface supports a number of buffers (defined at compile time), each of which supports a number of read ports and a write port. The gather-scatter interface currently supports a single write port per buffer, but could be modified to support more write ports in the future. The gather-scatter interface explicitly exposes an instruction buffer to the application kernel, and defines an instruction format (which includes space for application-defined opcodes) in order to support a generic preprocessing script that optimizes the data transfers for each work-list and statically detects data hazards.

Section 6.5 will discuss the implementation of the gather-scatter work-list, and includes a depiction of the double buffering and associated hazard avoidance logic.

4.2 Utility Interfaces

These interfaces do not specifically describe data structures, but instead describe useful functionality needed by many applications.

4.2.1 Scratchpad Interface

CoRAM++ control threads cannot directly read or write data in DRAM. While it is possible to detect and create logic for these accesses during compilation[81], this feature is not included in the CoRAM++ compiler in order expose the high latency of these accesses. The scratchpad agent allows control threads to access data in DRAM when explicitly required, and can prefetch data to take advantage of data access locality. This agent can also be used to manage memory-mapped accelerators directly attached to a DRAM interface.

Table 4.9 describes the control thread commands that the scratchpad interface supports. The first two functions are very simple, and do not attempt to take advantage of locality to

Control Thread Command	**Description**
thread_read (handle, addr)	Read 4 bytes from *addr* using scratchpad *handle*. Does not attempt to exploit locality.
thread_write (handle, addr, data)	Write *data* to *addr* using scratchpad *handle*. May read data from the global address space if the width of the scratchpad is more than 4 bytes. Does not attempt to exploit locality.
thread_read_cache (handle, addr,last_addr, offset, bytes, dirty)	Read 4 bytes from *addr* using scratchpad *handle*. The scratchpad is used as a cache holding *bytes* of data starting *offset* bytes from the start of the scratchpad, and reference parameter last_addr stores the address of the current item in the cache. *Addr* is aligned to *bytes* bytes, and if the aligned address does not match *last_addr*, a new cache line is loaded into the cache. If *dirty* is true, the data in the cache will be flushed to the current *last_addr* before loading the new data.
thread_write_cache (handle, addr, data, last_addr, offset, bytes, dirty, flush)	Write *data* to *addr* using scratchpad *handle*. he scratchpad is used as a cache holding *bytes* of data starting *offset* bytes from the start of the scratchpad, and reference parameter last_addr stores the address of the current item in the cache. *Addr* is aligned to *bytes* bytes, and if the aligned address does not match *last_addr*, a new cache line is loaded into the cache. If *dirty* is true, the data in the cache will be flushed to the current *last_addr* before loading the new data. If *flush* is true, then the cache will be flushed to the global address immediately after writing data to the scratchpad.
thread_flush_cache (handle, addr, offset, bytes)	Uses the Agent attached to *handle* to flush the cache of *bytes* bytes to global address *addr* starting *offset* bytes from the beginning of the scratchpad.

Table 4.9: Scratchpad application-level interface for control threads.

Control Thread Command	Description
read_from_host	Read signal from host system. The signal is a 64 bit token provided by the host system.
write_to_host	Write signal to host system. The signal is an arbitrary 64 bit token.

Table 4.10: Host computer application-level interface for control threads.

improve performance. The remaining three functions allow the scratchpad to be used as a sophisticated cache – the application can segment the scratchpad into multiple cache lines of application-selected size, and can specify for each cache operation whether the current cache line in the selected segment of the cache should be flushed back to the global address space if it will be replaced.

The linked list experiments in Section 7.3 show that the scratchpad can improve performance when there is locality in the application's data accesses, but can hurt performance when large caches are used and the application does not exhibit locality that can be exploited by the cache.

4.2.2 Host Computer Interface

The CoRAM++ host computer interface addresses one of the most vexing issues in FPGA development—getting data from a host computer to the FPGA and back. The agent supporting this interface transfers data from a host computer to FPGA dram, transfers computation results back to the host computer, provides run-time information to application components, and benchmarks execution through hardware counters. The host computer interface currently supports serial and Ethernet communication, but there is no reason why it could not use any communication interface on the FPGA.

API Type	API Port or Function	Description
Hardware Kernel Ports	notEmpty	Indicates whether or not a FIFO-style channel contains data.
	dout[]	The first data item in the a FIFO-style channel or the data in a register-style channel.
	deq	Dequeue from a FIFO-style channel.
	notFull	Indicates whether or not a FIFO-style channel has room for a new data item.
	din[]	Input data to write to the channel.
	enq	Write the data *din* to the channel.
Control Thread Commands	test_read_channel(c)	Returns true if channel *c* can be read.
	read_channel(c)	Read a data item from channel *c*, dequeueing if it is a FIFO-style channel.
	test_write_channel(c)	Returns true if channel *c* can be written.
	write_channel(c,v)	Write *v* to channel *c*.

Table 4.11: Channel application-level interface.

4.2.3 Channel Interface

The channel interface can be used by applications to send messages between any pair of application components—a hardware and a control thread, two hardware kernels, or two control threads. The channel interface creates a static link between components that must be defined at compile time. Table 4.11 shows the application-level interfaces available when using the CoRAM++ channel. The channel interface supports a pair of unidirectional interfaces—the register-style interface is a pair of registers each supporting a data flow in one direction, rather than a shared register that can be written and read by the components on either end of the channel.

Control Thread Command	Description
get_kernelcomm_write_addr(i)	Returns the address used to write to kernel-kernel communication interface *i*.
get_kernelcomm_read_addr(i)	Returns the address used to write to kernel-kernel communication interface *i*.

Table 4.12: Kernel-kernel communication interface for control threads

4.2.4 Kernel-Kernel Communication Interface

The channel interface discussed above can be used to statically define connections between two application components at compile time. The kernel-kernel communication interface allows dynamically-defined communication between hardware kernels.

The kernel-kernel interface operates by designating special addresses that can be used to transfer data between kernels. The application uses any of the application-level interface described above with these special addresses to communicate, meaning that any particular instances of the application-level interfaces can stream data from (or to) DRAM, an off-chip communication interface mapped into a global address space, or another kernel.

While it would be possible to use DRAM to transfer data between application hardware kernels, the kernel-kernel communication interface reduces the need for application components exchanging data to coordinate—the application must only ensure that the component that reads data does so at the same rate as the component that writes data.

This application-level interface supports streaming semantics—one application component writes data to the interface, and another application component reads from the interface.[2] It would also be possible to define a kernel-kernel communication interface

[2]The kernel-kernel communication interface does not actually enforce that different compoents use each side of the interface, but a single application component that both wrote and read data using this application-level interface would need to concern itself with implementation details such as available storage.

with random access rather than streaming semantics, but doing so would require more coordination between application components. Since the application defines both ends of the kernel-kernel communication—writing from one kernel and reading from another—defining random access semantics should not be required, since the purpose of this interface is to transfer data between local buffers attached to two application kernels.

4.2.5 Thread Creation Interface

The thread creation instantiates a control thread for the application. The application-level interface is embodied by a Verilog module that is parametrized by the name of the control thread function, an instance identifier (which is a number), and any method parameters to the control thread function. The CoRAM++ compiler ensures that all of the function's method parameters are present, and throws an exception at compile time if any are missing. The thread creation interface does not provide any run-time functionality to either control threads or hardware kernels, and is configured solely through its compile-time parameters.

Chapter 5

Inside the CoRAM++ Programming Environment

This chapter describes the implementation of the CoRAM++ programming environment in detail. It discusses the CoRAM++ compiler, which compiles application components, along with the agents supporting the application-level interfaces that are part of the application, to a set of Verilog files suitable for processing by FPGA vendor-specific tools. This chapter will also describe the interfaces used to create CoRAM++ agents.

5.1 The CoRAM++ Compiler

The CoRAM++ compiler builds CoRAM++ applications, processing application hardware kernels, control threads, and the included CoRAM++ agents into Verilog files ready for the vendor tools that generate FPGA programming files. The compiler can also build applications for simulation, and includes functionally accurate DRAM and communication models. During the compilation process, the CoRAM++ compiler uses a custom

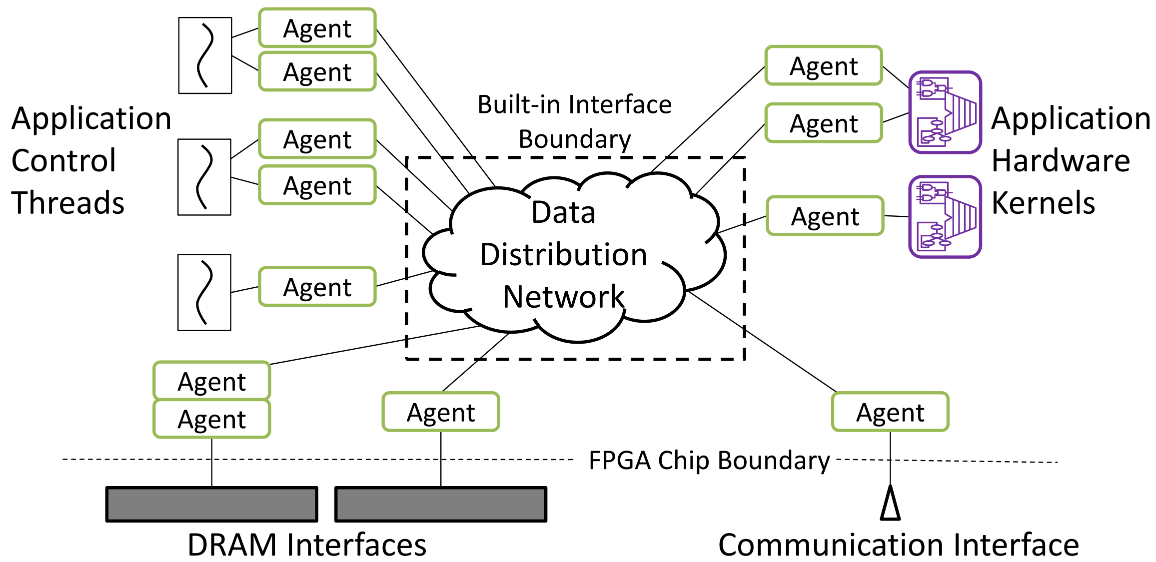


Figure 5.1: System-level view of a complete CoRAM++ application

LLVM [53] back end to compile control threads to Verilog finite state machines.

Figure 5.1 depicts a complete application, including application control threads and hardware kernels and DRAM and other communication interfaces that cross the FPGA chip boundary. Application components and interfaces are connected to CoRAM++ agents, which connect to each other using an internal data-distribution network. using a built-in programming interface. At the FPGA chip boundary, "accelerator" agents can connect directly to each DRAM interface or communication interface, allowing these "accelerator" agents low-latency interactions with the attached interface. Accelerator agents are generally data-structure-specific modules that make some pointer-chasing data access pattern as fast as possible. Multiple accelerator agents can be stacked in a pipeline at each DRAM interface, for example to accelerate a pointer-based data structure and support a general-purpose cache. Figure 5.1 only presents data connections supporting dynamic routing that go through the data distribution network—it omits the point-to-point connections created

by the internal communication channel interface.

The CoRAM++ compiler currently targets commodity FPGAs from both Altera and Xilinx. Applications and CoRAM++ agents are portable across different FPGAs, but can provide device-specific variants of each agent. Each compilation target includes a configuration file that specifies data path width, NoC parameters, and how DRAM controllers and communication interfaces are mapped into the address space. The CONNECT-based NoC supports any topology supported by CONNECT, and the CoRAM++ compiler can use ShrinkWrap [29] to reduce the width of the NoC and associated resource requirements. The output of the CoRAM++ Compiler is a set of verilog files and configuration information relating to routing for the NoC and the clusters.

The CoRAM++ compiler supports most t of the C language (as discussed in Section 5.2 below, but does not have built-in support for accessing memory from CoRAM++ control threads. CoRAM++ does not make any restrictions on hardware kernels other than that they use the application-level interfaces exported by CoRAM++ agents for communication. CoRAM++ hardware kernels can be created using any hardware design methodology or RTL language, including IP core generators and the high level synthesis tools discussed in Section 2. Much of the CoRAM++ infrastructure has been created in the Bluespec System Verilog [66] language, but application developers do not need to use Bluespec to create CoRAM++ applications.

The CoRAM++ Compiler was built on top of the soft-logic realization of the original CoRAM architecture [32], and consists of the following major components:

- The CoRAM++ control thread compiler, built using the LLVM compiler [53], which compiles CoRAM++ control threads and agent subthreads into Verilog state ma-

FPGA Board	Xilinx ML605	Terasic DE4	Digilent Zedboard	Xilinx ZC706
FPGA Chip	Xilinx LX240T [89]	Altera EP4SGX530 [15]	Xilinx XC7Z020 [91]	Xilinx XC7Z045 [91]
Hard CPU Cores	N/A	N/A	2× ARM Cortex A9	2× ARM Cortex A9
Logic Cells	241,152	531,200	85,000	350,000
Block Memory	1,872 KB	3,422 KB	560 KB	2,180 KB
DSPs	768	1,024	220	900
DRAM Bandwidth	6.4 GB/s	2×6.4 GB/s	4.3 GB/s	4.3 GB/s (Shared) 12.8 GB/s (Fabric-Only)
DRAM Capacity	512 MB	2×1 GB	512 MB	1 GB (Shared) 1 GB (Fabric-Only)
Host Interface	500 KBits/s Serial UART	115 KBits/s Serial UART	1 GBit/s Ethernet	1 GBit/s Ethernet

Table 5.1: Supported FPGA boards.

chines, and assembles the application based upon its specific communication needs.

- A system library containing the built in interfaces, written in Bluespec System Verilog [66]. This system library includes cluster, interface, and NoC components that are selected and assembled during compilation. The CoRAM++ built-in interfaces include all of the components of the original CoRAM architecture, which allows applications for the original CoRAM architecture to be compiled without modification. Section 5.3 describes the CoRAM++ built-in interfaces in detail.

- A build platform consisting of scripts that invoke the above two components, configuring the memory subsystem selected by the application, and assembling all of the output files for simulation or for generating FPGA programming files using vendor-specific tools.

Table 5.1 lists the currently supported FPGA boards. The CoRAM++ compile wrappers for floating point and other common IP cores (such as hard memory blocks) that are different between Xilinx and Altera devices. Both DRAM controllers on the DE4 are supported, and the mapping of the global address space to memory channels is configurable.

The Zynq ZC706 FPGA board includes two DRAM interfaces: a "PS" DRAM interface, which is shared between the ARM cores and reconfigurable fabric, and a higher-bandwidth "PL" DRAM interface, which is connected directly to the reconfigurable fabric. The CoRAM++ compiler target for this FPGA board supports both DRAM interfaces.

When targeting the ZedBoard, the CoRAM++ compiler removes the NoC and connects one or more CoRAM clusters directly to one or more of the Zynq's high performance AXI ports, reducing resource requirements on the small FPGA. The host computer agent runs primarily as software on one of the ARM cores, transferring data between the host computer and DRAM interface that is shared between the ARM cores and reconfigurable fabric, and uses a slave AXI port to signal the FPGA application. This variant of the host computer agent can be used to attach CoRAM++ applications to any AXI-based systems, and has been used to target LEAP. Applications are compiled to the Zynq platform without any application-level changes, and supporting high speed transfers to the board over Ethernet. Section 6.7 describes Zynq-specific details related to the ARM cores and communication with the host system.

The CoRAM++ compiler generates all of the artifacts necessary to create FPGA programming files using vendor-specific tools, including a per-target top level Verilog file instantiating DRAM controllers, clock generators, and application and CoRAM++ components. The CoRAM++ compiler supports Quartus 14.1 when targeting Altera devices,

and uses ISE 14.7 when targeting Xilinx devices. An effort to support Xilinx's Viviado suite is currently underway.

In addition to targeting FPGA devices, CoRAM++ applications can be simulated with several RTL simulators, and has been validated with iVerilog, an open source RTL simulator, Xilinx ISim, and Synopsis VCS. ISim is currently the preferred simulator, even though it is slower than VCS, because it is straightforward to incorporate with IP blocks for floating point and DRAM interfaces when using the ISim simulator. The simulation environment supports several memory models, including:

- A simulated DRAM subsystem that assumes that every request takes a fixed number of cycles. The exact number of cycles is configurable, and this subsystem can easily be initialized at start up and saved to disk at simulation end for validation.
- The Xilinx Bus Functional Model for the Zynq processor. This interface simulates the Zynq processor memory subsystem, and supports the Zynq processors in the same way as the ZedBoard. This model does not support simulating the ARM processor, but does simulate AXI bus transactions.

5.2 CoRAM++ Control Thread Capabilities and Limitations

CoRAM++ control threads simplify the specification of global control and data transfers as much as possible, and are intended to preside over long running operations that whose management is not the bottleneck for application performance. As such, control threads can be compiled from C-language source code to effective Verilog state machines without the need for automatic data parallelization in order to achieve good performance.

Control threads are specified by C-language source code that is processed by the LLVM

compiler, and as such supports all typical C-language constructs, including if statements, loops, and function calls. Control Threads can use all of the integral data types supported by LLVM 2.8, and all operations on integral data types. However, application developers should be aware of the fact that multiplication is expensive when the multiplicand or multiplier are not a power of two, and similarly, that division is expensive when the divisor is not a power of two. In general, control threads should limit computation to address generation and control flow, but can perform other computations when the speed of these computations is not important for performance.

Control threads can use structures and arrays whose size is fixed at compile time, but cannot use variable size arrays. Control threads cannot use floating point numbers, and cannot directly access data in DRAM without using the Scratchpad agent (discussed in Section 4.2.1). The C-to-CoRAM compiler [81] demonstrated that it is possible to detect these accesses and automatically instantiates the structures needed to perform them, but the CoRAM++ compiler omits automatic scratchpad instantiation due to the high latencies of these accesses.

Control threads can define functions, including recursive functions, but should avoid recursion and declare functions as "inline" where possible. CoRAM++ creates a fixed-depth stack that is used by function calls that cannot be inlined. The depth of the stack is specified within the target-specific configuration file (which can be customized by the application developer). The best practice is for all Agent API functions to be specified as "inline," which will reduce the need for the stack.

Each command port supplied by CoRAM++ agents can only connect to a single control thread. While the CoRAM++ programming environment allows agents to expose multiple

command interfaces (which is intended to allow communication channels between control threads), CoRAM++ does not provide any direct support for interleaving commands from multiple control threads into a single command stream. Any CoRAM++ agents that expose multiple command interfaces would need to resolve multiple command streams themselves.

5.3 CoRAM++ Built-in Interfaces

The CoRAM++ compiler directly supports a small set of built-in interfaces that support low-level functionality for Agents, which can be created without modifying the CoRAM++ compiler or supporting infrastructure. These built-in interfaces mae up the system layer discussed in Section 1.1. CoRAM++ applications can use these built-in interfaces directly, but should use the application-level interfaces exposed by CoRAM++ agents when possible. CoRAM++ agents are created in the CoRAM++ programming environment, which means that they can use the application-level interfaces exposed by other CoRAM++ agents in addition to the built-in interfaces described in this section. In general, the hardware components of CoRAM++ agents can be created using RTL language, but accelerator agents that are directly connected to a DRAM or communication interface must currently be created in Bluespec.

5.3.1 Block Copy Interface

The block copy built-in interface performs the bulk of the work of data transfers, and is equivalent to CoRAM blocks in the original CoRAM architecture. This interface supports the same ports and function calls as does the block copy application-level interface, which is described in Table 4.1, and supports either one or two SRAM-style ports on the hardware

side of the interface. The interface supports compile-time parameters for the width, depth of interface.

Instances of the block copy interface use modules called "clusters" to connect to the NoC and control threads, which work the same way as clusters in the soft-logic realization of the original CoRAM architecture [32]. Clusters serve as NoC endpoints for CoRAM++ agents, and map DRAM controllers and communication interfaces into a global address space.

5.3.2 Channel Interface

The built-in channel interface is the same as the channel application-level interface described in Section 4.2.3.

5.3.3 Thread Creation Interface

The built-in thread creation interface is the same as the thread creation application-level interface described in Section 4.2.5. This built-in interface allows agents to instantiate their own control threads, which is important for allowing agents to provide the most convenient possible application-level interfaces, as is described in Appending 3.2.

5.3.4 Script Execution Interface

The script execution interface is used to run a parameterized script at run time, primarily to allow agents to run an IP core generator with parameters specific to the way that the application instantiates the agent. For example, the multi-dimensional DFT application discussed in Section 7.2 uses this interface to invoke the Spiral streaming permutation and Discrete Fourier Transform generators [60], and the CoRAM++ web-based application builder [1] can use this interface to invoke the Bluespec [66] compiler when applications

instantiate agents written in the Bluespec language. The CoRAM++ web-based application builder prevents users from directly instantiating the script executor interface for security reasons.

5.3.5 Control Thread System Library Interface

CoRAM++ includes a system library interface supporting the functions listed in Table 5.2, which includes all of the functions of the original CoRAM architecture for backwards compatibility. These functions are supplemented by application-level interfaces exported by the CoRAM++ agents instantiated by the application, and constants that are defined by a include files that are shared between the hardware kernels and control threads.

5.4 CoRAM++ Accelerator Agents and Software Accelerators

CoRAM++ defines high level data-structure-specific application-level interfaces backed up by agents implementing the desired behaviors. The application is unaware of the implementation details of each agent, which can instantiate logic anywhere within the FPGA to support the public interface, and can even use software running on tightly coupled hard-logic CPU cores to accelerate irregular pointer-chasing data accesses.

CoRAM++ accelerator agents are the primary mechanism for making pointer-chasing memory accesses run faster, and are hardware modules that are attached directly to a DRAM interface. Attaching accelerator agents directly to a DRAM interface minimizes the latency between the accelerator module and the DRAM interface. CoRAM++ accelerator agents can also be connected to a communication interface to support device-specific operations using that interface.

Each accelerator agent is parameterized and interacts with the attached interface through

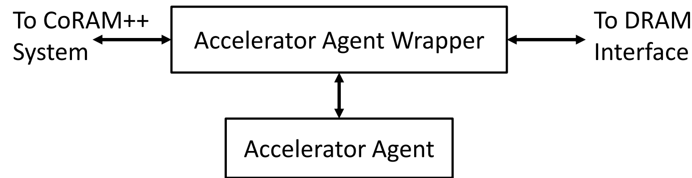


Figure 5.2: Accelerator Agent Wrapper Module

a thin portability layer. When attached to a DRAM interface, accelerator agents operate on requests that match the width of the attached DRAM interface, and map themselves into the address space of the attached DRAM interface by intercepting memory requests addressed to the DRAM interface. Since accelerator agents map themselves into the address space of the attached DRAM interface, they can define their own commands by defining particular writes and reads to the mapped address space. For example, a linked list accelerator agent can define a write to a particular address as an operation that initiates traversal of a linked list, using the written value as the start address for linked list traversal. A different address in the memory-mapped address space might be used to merge a pair of linked lists.

The portability layer defined by CoRAM++ and used by accelerator agents specifies that accelerator agents must deliver read responses in the order in which requests were issued, and specifies that there are no responses to write requests. Multiple accelerator agents can be attached to the same DRAM interface in a pipeline between the CoRAM++ NoC and the attached DRAM interface.

The CoRAM++ compiler includes an accelerator agent wrapper module that helps accelerator agents meet ordering requirements when interleaving requests that the accelerator agent handles with requests that it does not. Figure 5.2 shows how this wrapper attaches to the interface, NoC endpoint, and Interface Agent. This module can optionally insert

pipeline stages at its input from the NoC and output to the attached interface in order to help the application meet timing requirements.

Accelerator Agents must currently be created in Bluespec, although it is possible to support Verilog Accelerator Agents in the future through a Bluespec shim. Table 5.3 presents the interface that must be exposed by accelerator agents, which use standard client/server semantics. Accelerator Aaents operate on *request* objects containing an address, write data, data mask bits, read/write flag, and transaction and packet tags. Accelerator agents operate on read *response* objects containing read data, data mask bits, and tags.

The CoRAM++ compiler does not directly include interfaces for instantiating accelerator agents, but does support compile-time hooks to allow accelerator agents to be attached at any DRAM interface or communication interface. Future work will support a cleaner, automated interface for instantiating accelerator agents.

CoRAM++ software accelerators perform the same function as accelerator agents, but are software modules that run on a hard-logic CPU core within a tightly coupled CPU-FPGA system, operating on the principle that the higher clock speeds of hard-logic processor cores and caches can interact more quickly with DRAM, and are also easier to write. Section 7.3.2 demonstrates the effectiveness of software accelerators on currently available systems.

5.5 CoRAM++ Cache

CoRAM++ includes a cache that applications can enable or disable at compile time. The cache is transparent to the application, attaches directly to the NoC, and can improve the performance of unstructured accesses that exhibit locality. The cache is optional except

when the width of the NoC is not configured to match the width of the DRAM Interface, which can reduce NoC resource requirements while also reducing available bandwidth. However, streaming applications that do need all available full DRAM bandwidth will configure the NoC width to match the DRAM interface width, and in general will elect not to use the cache.

Function Name	Description
cpi_register_thread	Control threads must call this function register themselves as control threads. Control threads in the original CoRAM architecture called this function to both register themselves and declare the number of instances of the function to create. In CoRAM++, control threads can also 0 as the number of instances to create, and use the built-in control thread creation interface (see in Section 5.3) to instantiate control threads with particular parameters.
cpi_instance	Control Threads can use this function to determine which instance of the control thread they are, for cases when different instances of a thread must perform different actions. This function exists for primarily backwards compatibility with applications for the original CoRAM architecture, as CoRAM++ applications can use the Thread Creator agent to vary the parameters that are passed to different instances of a Control Thread.
cpi_split	When compiling Control Threads, the CoRAM++ compiler creates a single state for every LLVM basic block in the each Control Thread, possibly chaining together a sequence of operations producing a single result. If the chain of operations is too long, it can negatively affect the clock speed of the application. This function can be used to split a basic block into two, breaking a chain of operations into two clock cycles with an intermediate register.
cpi_noop	This function can be used to prevent loops from being optimized away.
cpi_time	This simulation-only function can be used to determine how many Control Thread cycles have elapsed since the simulation began.
cpi_dumpmem	This is a simulation-only function that saves the contents of simulated DRAM to disk. The function uses functionality built into the simulator in use (ISim, VCS, etc) to actuall save data, so the format of the data is dependent on the simulator.
cpi_printf	This is a simulation-only function that can be used to print information for debugging. It supports the same string formatting functions as the simulator in use.
cpi_finish	This simulation-only function is used to stop a simulation, making it easier for the user to run a complete simulation without knowing how many cycles the simulation will take.

Table 5.2: CoRAM++ system library for control threads

Bluespec API Function	**Description**
usr.request.put(request)	Requests from the application come through this function.
usr.request.accept	The accelerator agent must return **true** if the accelerator agent accepts the request provided through a call to *usr.request.put* this cycle. If the accelerator agent does not return **true**, it will receive the request again in the next cycle. The accelerator agent should return **false** if there was no call to *usr.put* this cycle.
usr.response.rdy	The accelerator agent must return **true** if it can provide a valid read response this cycle, and **false** otherwise.
usr.response.get	The accelerator agent returns a read response.
usr.response.accept	The system calls this function to inform the accelerator agent that the response returned by *usr.get* this cycle has been accepted.
mem.request.rdy	The accelerator agent must return **true** when it is making a request of the attached interface.
mem.request.get	The accelerator agent returns its current request to the attached interface.
mem.request.accept	The system calls this function to inform the accelerator agent that the request provided by the accelerator agent this cycle has been accepted.
mem.response.put(response)	The system calls this function to provide a read response from the attached interface.
mem.response.accept	The accelerator agent must call this function to inform the system that it accepts the response supplied to the Agent this cycle.
agentHandlesRequest(request)	The accelerator agent must return **true** if the accelerator agent will process *request*, and false if the accelerator agent wrapper should forward the request directly to the attached interface. This method is only required if the accelerator agent wrapper is used, and the accelerator agent should return a constant **true** if the accelerator agent wrapper is not in use.

Table 5.3: Interface that accelerator agents must support.

Chapter 6

Constructing the CoRAM++ Agents

This chapter shows how CoRAM++ Agents are constructed out of built-in interfaces, and how this construction allows CoRAM++ agents to provide convenient application-level interfaces to applications. The CoRAM++ agents are designed to support "fire-and-forget" application level interfaces that allow the application control thread to invoke a command without needing to account for the fact that the command may take a long time to complete. Applications can query the status of commands if necessary, but in general will invoke a sequence of long-running commands and only wait for the final command to complete. Section 3.2 shows why it is not trivial to build application-level interfaces that operate in this "fire-and-forget" fashion. CoRAM++ agents typically support "fire-and-forget" application-level interfaces by incorporating an control thread within the agent that can independently manage blocking data transfers on behalf of an application control thread without blocking the application control thread.

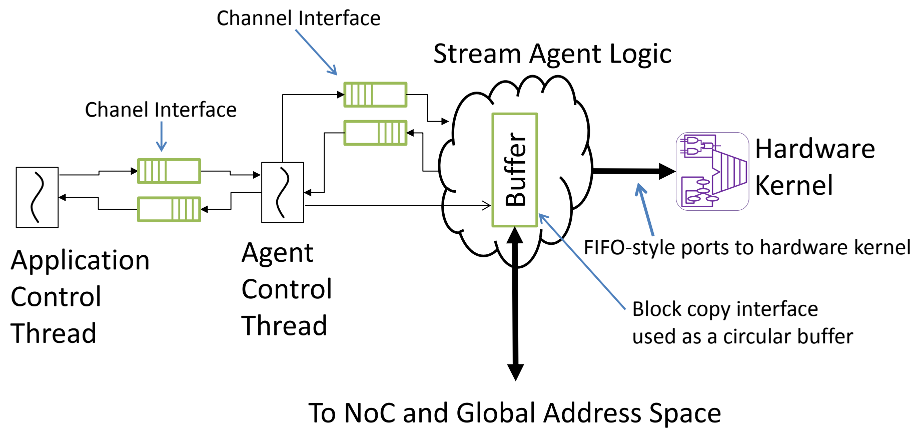


Figure 6.1: Stream agent diagram, showing the read stream agent (the write stream agent is almost identical). All modules except for the control thread and hardware kernel are instantiated by the stream agent.

6.1 Stream Agent

The goal of the stream agent is to provide the most convenient possible "fire-and-forget" application-level interface for streaming accesses. The stream agent supports the application-level interface described in Table 4.2 for a variant of the agent that streams data to a hardware kernel, and supports the application-level interface described in Table 4.3 for a variant of the agent that streams data from a hardware kernel. While the stream agent uses a circular buffer to support streaming data transfers, it includes a separate agent thread (compiled to a separate FSM by the CoRAM++ compiler) to mask the need for flow control when transferring a large amount of data through the circular buffer.

Figure 6.1 shows the how the of the stream agent is constructed. The application's control thread connects to a built-in channel interface to send commands to a control-thread instantiated by the agent, and the application's hardware kernel connects to wire-level ports providing a FIFO-style interface. The agent's control thread uses the built-in

block copy interface to perform data transfers, and incorporates head and tail pointers to implement a circular buffer. The agent's control thread can accept commands from the application's control thread without blocking the application control thread, even though the agent's control thread might need to block due to flow control requirements.

6.2 Multi-Dimensional Array Agent

Figure 6.2 shows how the multi-dimensional array agent is constructed. This agent is similar to the stream agent in that it uses the built-in block copy interface for data transport, an agent control thread to manage data transfers, interacting with the application control thread through the built-in channel interface. The multi-dimensional array agent uses a modified version of the stream agent which changes the source code of the agent's control thread. The multi-dimensional array agent's control thread includes support for the address generation requirements of multi-dimensional data transfers.

The figure also shows how the application-specified stream permutation engines are instantiated between the incoming data stream and the application's hardware kernel. The application can, at compile time, configure the multi-dimensional array agent to move data across clock domains while retaining full bandwidth—for example moving data from NoC data path that is 64-bytes wide and runs at 100 MHz to one that is 32 bytes wide but runs at 200 MHz. This allows the multi-dimensional array agent to support high-speed IP cores while retaining the easier routing of low-speed data paths.

6.3 Linked List Agent

The linked list agent traverses linked lists, allowing applications to read or write data payloads associated with the linked list, or merges two sorted linked lists in place. The

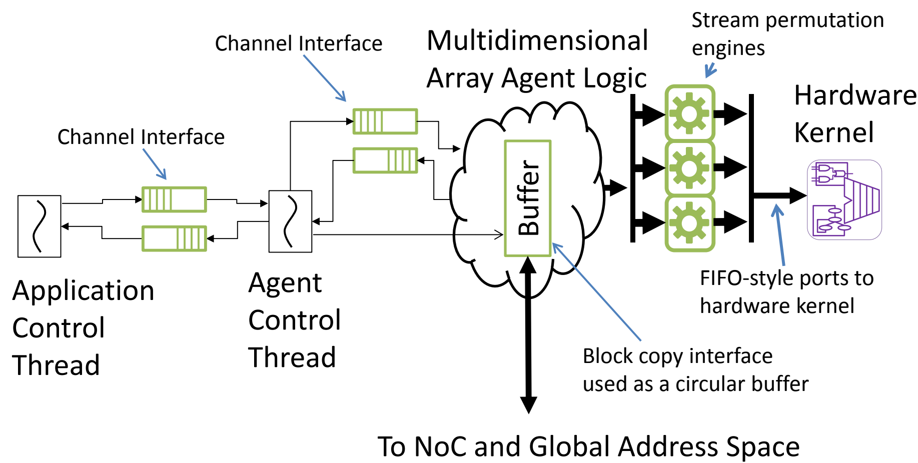


Figure 6.2: Multi-dimensional array agent diagram, showing the variant of the agent that allows the application hardware kernel to read data. This agent is similar to the stream agent, but supports multi-dimensional array data, and can stream individual data elements, elements that are sequential along any dimension, or multi-dimensional tiles of data. The agent supports row-major or tiled data layouts.

linked list agent requires that the format of the linked list nodes be defined at compile time, and supports linked lists with data payloads stored within the linked lists nodes as well as linked lists with pointers to data payloads in the linked list nodes.

The core functionality of the linked list agent is supported by a linked list hardware engine, which is depicted in Figure 6.3. This hardware engine is configured with the data format of the linked list in use, and has interface ports for requesting data from DRAM, writing data to DRAM, and sending data to the application. The hardware engine includes a small direct mapped cache, typically configured to store 4 words of data. The data width of the cache is also configurable, and typically matches the native DRAM interface width. The hardware engine contains a sub-component used for merging linked lists, and can either be instantiated within the hardware kernel portion of the agent or as an accelerator agent.

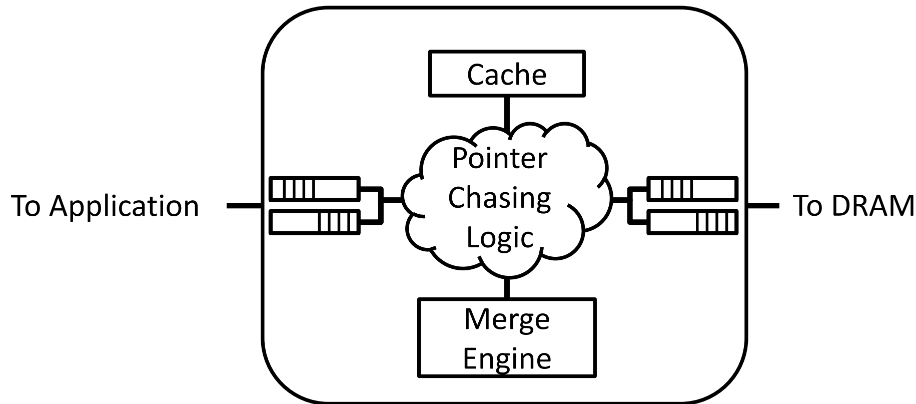


Figure 6.3: Linked list engine, including an embedded linked list merge engine and a small cache. When not instantiated as an accelerator agent, the port labeled "To DRAM" is connected to a pair of stream agents.

When instantiated as an accelerator agent, the hardware linked list engine maps itself into the address space of the interface to which it is attached by processing all addresses with an application-specified address bit set to 1, for example if the application specifies that the Agent uses bit 29, then the Agent will capture all accesses to addresses starting at 0x2000000. Since the bit to use is a parameter, multiple instances of the Agent can attach to different parts of the address space. Note that the number of address bits used is configured in the board-specific configuration file and defaults to 32 bits, allowing applications to increase the size of the global address space if necessary.

The hardware engine is controlled by a simple command/data signaling mechanism: All requests sent to the lowest address in the Agent's address space are treated as commands, and all other requests are treated as data requests. For example, using the start address of 0x20000000 as discussed above, the application will write to address 0x20000000 to initiate a linked list operation, and read from this address to query for available data. Once the application is aware that data is available, it can initiate a large streaming ac-

cess to the first valid address above 0x20000000 (which depends on the board-specific configuration data) for linked list data.

The linked list agent generally instantiates a pair of stream interfaces, one for reading from DRAM and one for writing to DRAM, and also can include a scratchpad agent for controlling the hardware engine when the hardware engine is instantiated as an accelerator agent. Since an application developer may also be using stream agents and a scratchpad interface for other purposes, the linked list agent includes application-level interfaces for controlling the linked list hardware engine accelerator agent without instantiating a second scratchpad agent in order to reduce resource requirements. The linked list agent can also omit the instantiation of its internal stream interfaces when this behavior is requested by the application.

We've also created a software accelerator version of the linked list accelerator. This software accelerator supports the same features as the hardware linked list engine. For maximum flexibility, the linked list traversal function takes a function pointer to be called with each linked list element, which can be used to either debug traversal or transfer information to the FPGA fabric. During testing, the callback functions used during traversal were always annotated as inline, and the test programs were disassembled to verify that invocations of the callback functions were actually inlined during compilation. Section 7.3.2 shows how we tested the software accelerator for linked lists, including streaming data to the FPGA fabric directly through dedicated AXI interfaces and through memory-based buffers.

6.4 B-Tree Agent

The B-Tree agent is very similar in structure to the linked list agent, in that it includes a B-Tree-specific hardware engine. The hardware engine can be configured for various B-Tree node widths, and which is used to calculate the number of keys (and child pointers or payload data entries) stored in each tree node. The B-Tree agent can be configured to cache the root node as in order to limit the number of memory accesses that must be performed when traversing the tree. In the future, this agent might be updated to cache multiple levels of the B-Tree, but the initial implementation of the this agent avoids doing so due to the complexity of flushing many tree nodes to memory when splitting the root node. A multi-level B-tree cache would likely be implemented using a pipelined Dynamic Search Tree [95].

6.5 Gather-Scatter Work-List Agent

The gather-scatter work-list agent operates on application that can be broken into sub-tasks, and executes each sub-task in sequence. The gather-scatter work-list agent uses a gather-list (a sequence of variable-sized operations that follow pointers to gather data) to transfer data to each application buffer, then invokes the application kernel, and transfers modified buffers back to memory, using the gather-list from the modified buffer as a scatter list.

Figure 6.4 shows the structure of the gather-scatter work-list agent, when configured with two sets of application buffers. The agent uses the scratchpad agent to load each gather-list and scatter-list, which are transferred using an instance of the stream agent for data moving in each direction. The gather-scatter work-list interface defines a standard

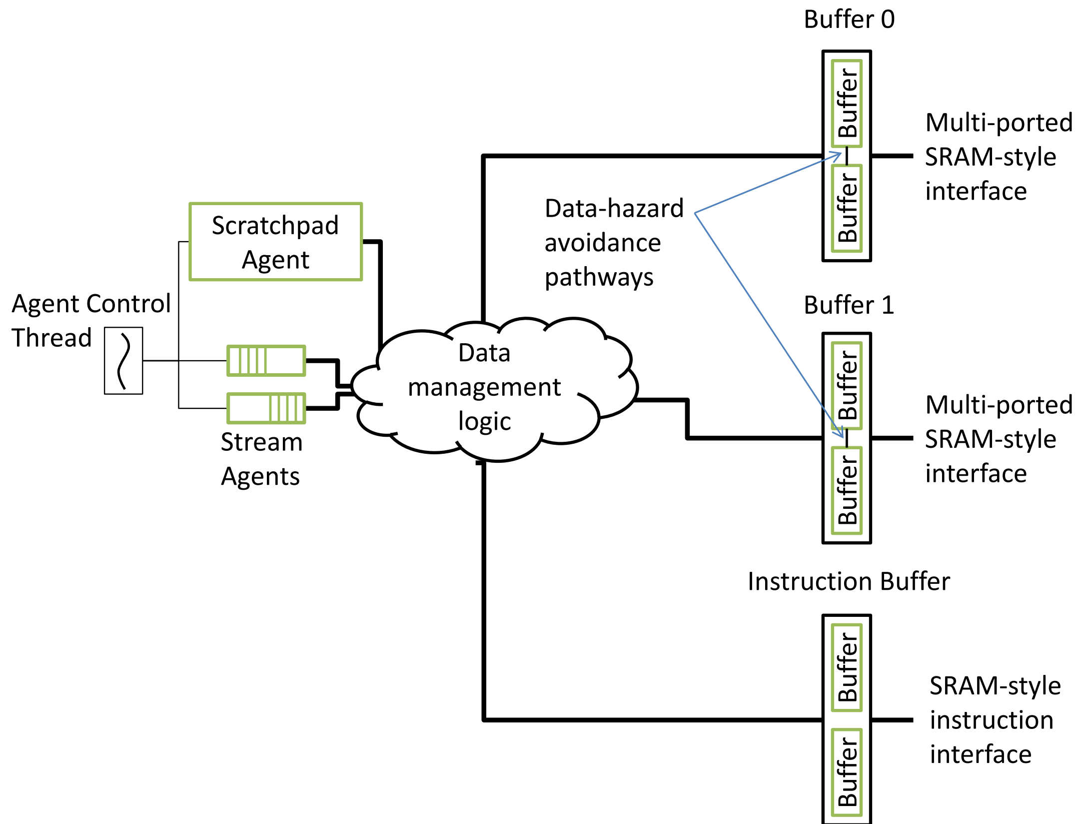


Figure 6.4: Gather-scatter work-list agent diagram, when the agent is configured with two application buffers

instruction format (which includes space for an application-specific opcode) in order to allow the interface to provide standard sub-task processing scripts to optimize data transfers, schedule double-buffered computation, and statically detect and avoid data hazards between sub-tasks.

The different options for optimizing the data transfers associated with each sub-task are:

1. Scanning through the gather-list for each application buffer, and coalescing data

transfers for adjacent items in the subgraph. Instructions are always stored sequentially, and are always transfered in a single block, but application data cannot always be transferred sequentially because each data item may be part of multiple sub-tasks.

2. Sorting the data items in each gather list to increase opportunities to coalesce data transfers. Since the agent supports a pre-defined instruction format and instructions that application data by local buffer address, each instruction can be re-written to account for the fact that this sorting process rearranges data within the local buffers.
3. Sorting the data items in each gather list and filling in small gaps in order to further coalesce data transfers.

There is never any reason not to apply the first two optimizations, since they do not increase the amount of data transferred when executing the each sub-task, and make data transfers more efficient. In practice, the third optimization is also advantageous as it does not greatly increase the amount of data transferred, and the advantages of coalescing data transfers are greater than the disadvantages of transferring a few extra data items the data fits in the local buffers. The preprocessor aborts the third optimization if it would transfer too many data items to fit in the buffers. Initial testing on the Xilinx ML605 indicated that these optimizations were necessary to saturate the computation engine on even simple computations, and Section 7.4 shows that these optimizations allow the gather-scatter work-list agent to saturate even the higher bandwidth of the Altera DE4 FPGA board.

6.6 Scratchpad Agent

Figure 6.5 shows the construction of the scratchpad agent, which retrieves DRAM data for use by control threads. The scratchpad agent uses the built-in block copy interface to

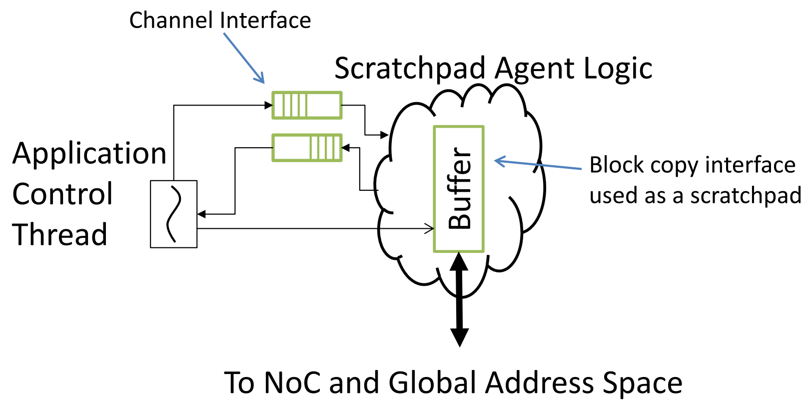


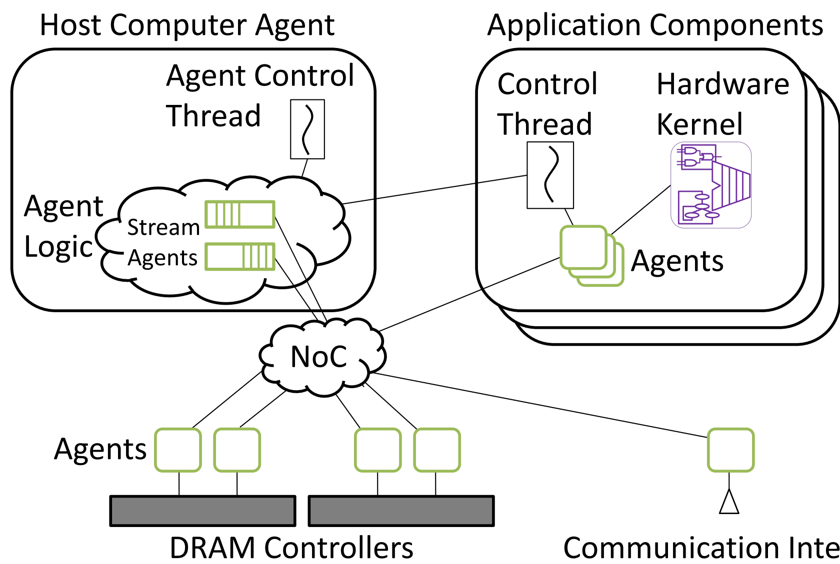
Figure 6.5: Scratchpad agent diagram. The block copy interface is used to buffer data locally, which is transferred to the attached control thread by the channel interface.

retrieve data, which is transferred to the control thread using the built-in channel interface. The agent includes functions that can be used to prefetch blocks of data from the global address space in order to exploit data locality in data accesses. The scratchpad agent is parameterized by the width and depth of the scratchpad, and its width should generally match the width of the NoC and DRAM interface for maximal performance.

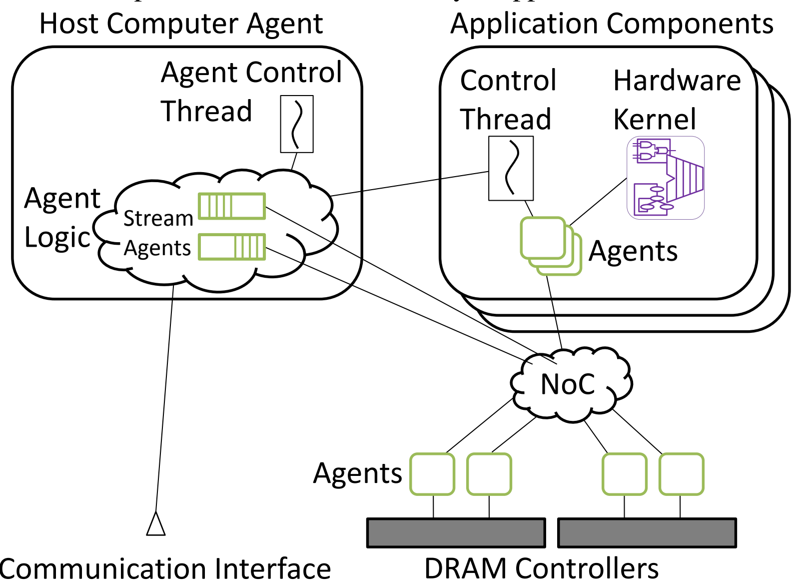
6.7 Host Computer Agent

The host computer agent supports several configurations that instantiate different components depending on the FPGA in use. Figure 6.6 shows the configurations that are available on FPGAs that are not using an embedded processor core, and Figure 6.7 shows the configurations that are available on "System-on-Chip" (SoC) FPGAs that contain embedded processor cores and AXI buses.

Figure 6.6 depicts an FPGA system which does not contain hard processor cores, such as the Xilinx ML605 and Terasic DE4 boards. The configuration in Subfigure 6.6a maps the communication interface into the global address space, allowing its use by any

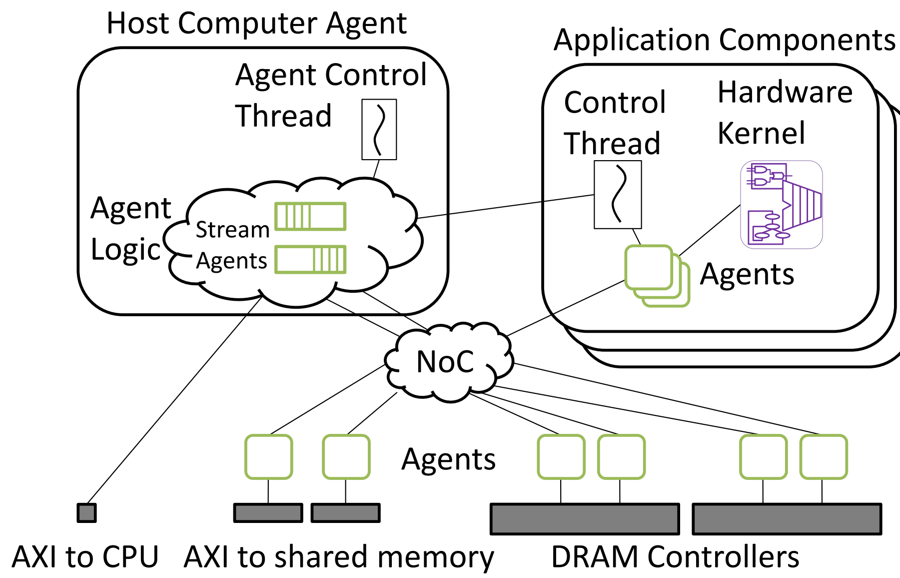


(a) FPGA with no processor cores and memory-mapped communication interface.

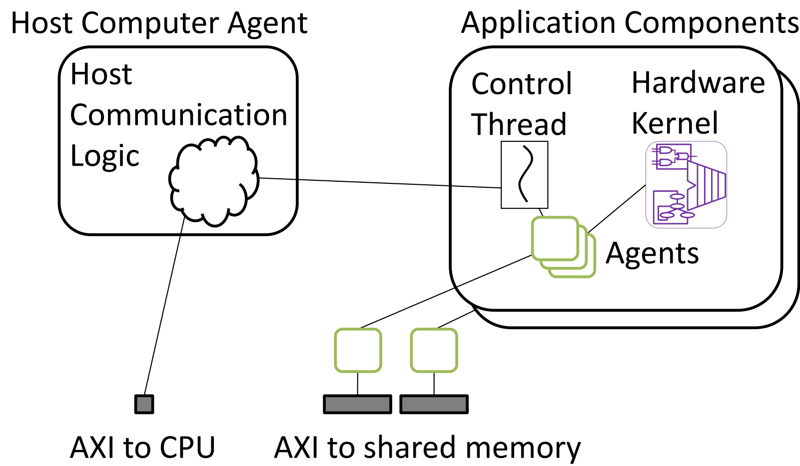


(b) FPGA with no processor cores and directly-connected communication interface.

Figure 6.6: Host computer agent diagrams for FPGAs with no processor cores, mapping the communication interface into the global address space a or connecting the communication interface directly to the host computer agent b.



(a) Large SoC FPGA



(b) Small SoC FPGA.

Figure 6.7: Host computer agent diagrams for System-on-Chip FPGAs with embedded processor cores. Large System-on-Chip FPGAs have both shared and fabric-only memory controllers a, and small System-on-Chip FPGAs only contain shared memory controllers b.

CoRAM++ agents that can access global addresses. The host computer agent is configured to include a pair of stream agents to read data from the communication interface and write

to DRAM or vice versa. Since most applications will not need to interact with the communication interface except through the host communication interface (supported by the host communication agent), the configuration in Figure 6.6b connects the communication interface directly to the host communication agent, which reduces resource requirements by removing the NoC endpoint that connects to the communication interface in Figure 6.6a.

The host computer agent also supports different configurations for SoC-style FPGAs, depending on whether the FPGA contains a DRAM interface that is not shared with the embedded processor cores. SoC-style FPGAs typically include an AXI bus that is used to connect the embedded processor to the reconfigurable fabric, and typically will also include at least one DRAM interface that is shared between the embedded processor and reconfigurable fabric. Subfigure 6.7a depicts a the configuration of the host computer agent on a "large" SoC FPGA containing shared and fabric-only DRAM interfaces, such as the Xilinx Zynq ZC706. In this type of SoC FPGA, the host computer agent connects to an AXI slave port, and connects both the shared and fabric-only DRAM interfaces to the CoRAM++ NoC. A small program running on the ARM core mediates between the host computer and CoRAM++ components.

Subfigure 6.7b depicts a "small" SoC FPGA, which contains a shared memory but no additional memory controllers, such as FPGA on the Digilent ZedBoard. Since these FPGAs tend to contain a small reconfigurable fabric, it is important to reduce resource requirements as much as possible. As such, CoRAM++ will typically remove CONNECT-based soft-logic NoC, and attach the CoRAM clusters that perform data transfers directly to one or more AXI ports. The host computer agent instantiates logic to attach to the ARM cores' AXI port, but does not need to move data to the shared DRAM controller.

Listings 4 and 5 show that effort involved in integrating the host computer interface (and its supporting agent) into an application is minimal. The application need only instantiate the host computer agent within its hardware kernel, and interact with the application-level interface exposed by the agent through function calls exported by the agent. The application can then read from this channel to obtain a signal to start, which indicates both that the user has initialized DRAM and that the application should use the run-time parameter provided with the signal. When the application is done, it should send a result code (which can include debugging information) to the channel. The "no-op" round trip time of this communication has been measured at 12 clock cycles at 100 MHz.

The host computer agent uses the same communication protocol regardless of the physical connection in use, which is described in Appendix B describes this communication protocol in detail.

6.8 Kernel-Kernel Communication Agent

The kernel-kernel communication agent allows dynamic routing of data between any pair of application kernels at the full bandwidth available by the NoC in use. The NoC topology created by CoRAM++ is bipartite, logically placing DRAM interfaces on one side of the NoC and application components on the other, regardless of the physical topology that is generated (which is controlled by the application developer). In order to allow arbitrary and dynamic routing at full speed, each instance the kernel-kernel communication agent creates a pair of NoC endpoints, one for writing and the other for reading.

Figure 6.8 shows an example application with these "agent NoC endpoints", demonstrating the two extra NoC endpoints with a FIFO in between them. It would also be

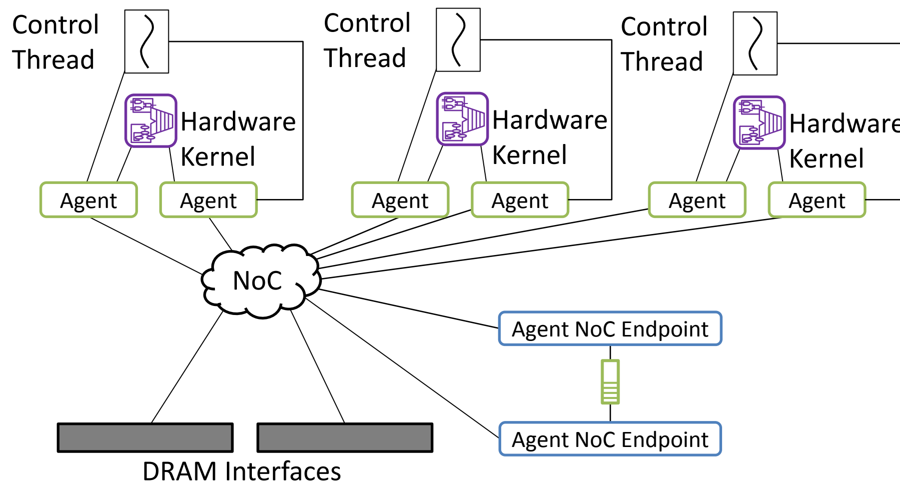


Figure 6.8: Kernel-kernel communication agent diagram, showing how this agent instantiates "agent NoC endpoints" to support dynamic kernel-kernel communication at full NoC bandwidth

possible to create some sort of shared local memory between these endpoints, but doing so would likely require more coordination between application components. The intended use model of this agent is streaming data between kernels (which can be stored in local buffers any way the application desires). Each NoC endpoint receives a unique block of the global address space (more than a single address to allow large data streams to be specified through a single control thread command), and the application can dynamically specify communication by using the appropriate addresses.

Chapter 7

Evaluating the CoRAM++ Programming Environment

Our evaluation of the CoRAM++ programming environment seeks to demonstrate that CoRAM++ provides convenient application-level interfaces without introducing undue resource or run time overheads.

7.1 Streaming Accesses

These experiments evaluate the best case performance of the CoRAM++ programming environment, and show the overhead of CoRAM++ when attempting to achieve this best case performance. DRAM interfaces generally deliver their best performance when performing large sequential accesses. These experiments stream sequential data from one of the DE4's DRAM controllers, through a streaming 64 input double precision DFT (generated by Spiral [61]), and to the board's other DRAM controller, built from source code similar to the example in Chapter 3.5. These experiments evaluate a DFT kernel running

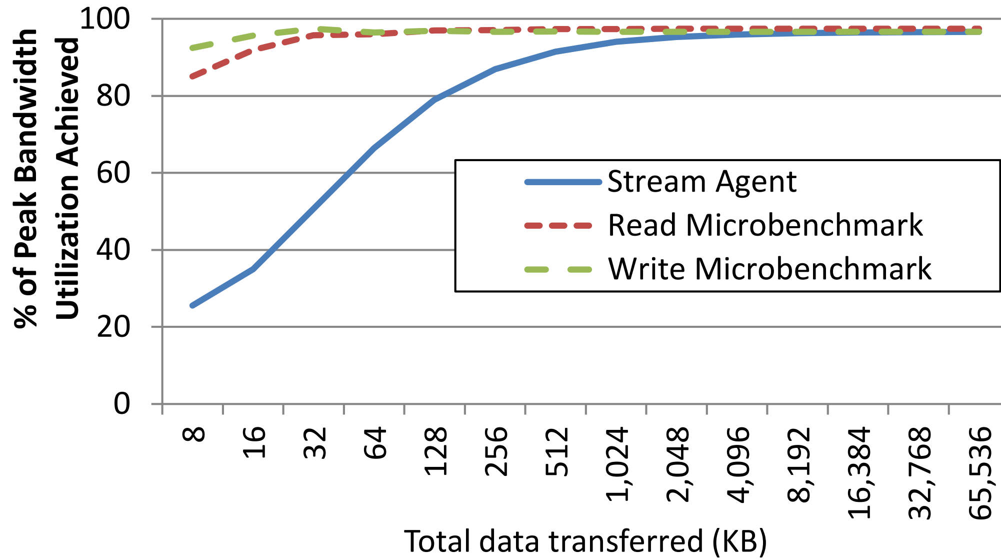


Figure 7.1: Streaming double precision 64-input DFT performance and DRAM controller microbenchmark results on the DE4.

at 200 MHz, operating on two inputs per cycle, matching the DRAM interface width on the DE4.

Figure 7.1 presents our experimental results. Data streams varied in size from 8 kilobytes to 64 megabytes. The figure also includes read and write microbenchmarks, which stream large amounts of data to or from sequential addresses in DRAM, and demonstrate the best real-world performance that can be achieved by the DRAM interfaces on the DE4. These microbenchmarks show that the DE4 can achieve up to 97% of peak bandwidth when reading, and over 96% of peak bandwidth when writing.

The CoRAM++ benchmark achieved approximately 90% of peak DRAM bandwidth on the DE4 when transferring 512 kilobytes of data, and matched the microbenchmark performance when transferring at least 8 megabytes of data, which shows that the runtime overheads of the CoRAM++ streaming application-level interface does not prevent

applications from saturating an application's DRAM interface. Performance on smaller data sets was limited by the small data set size and 242-cycle latency of the DFT core. Since the 8 kilobyte data transfer required 256 cycles due to the data width of the DE4's DRAM interfaces, the latency of the DFT core almost doubled the minimum computation time.

The streaming DFT used approximately 32% of the logic and 7% of the block memory available on the DE4. Table 7.1 breaks this resource utilization down by component, showing that the DRAM controllers and NoC consume significant resources. An FPGA (such as [6]) that hardens these components would incur less overhead. ShrinkWrap [29] could also reduce NoC-related resource requirements for application scenarios that are less bandwidth constrained.

7.2 Multi-Dimensional Array Accesses

This application scenario shows that CoRAM++ can provide applications with convenient use of more complex data structures, while giving the application developer choices about data structure traversal and data layout in DRAM that affect application performance. We used the the multi-dimensional array agent, which allows the application developer to select array traversal order and data layout at run time. Data may be stored in

Table 7.1: CoRAM++ 1D DFT resource breakdown by component.

Component	**ALUT %**	**Register %**	**Block Mem %**
DRAM Controllers	11.4	12.3	15.3
NoC	6.7	6.2	0
NoC Endpoints	16.3	23.8	25.6
Threads + Agents	9.0	12.8	44.9
Kernel	56.6	44.9	14.2

the common row-major order, or may be stored in a tiled order. Data stored in tiled order can be traversed quickly along any array dimension. The multi-dimensional array agent can also be configured with a set of data permutation engines, which can be activated at run time to reorder data. In these experiments, we used permutation engines to recover rows from tiles of data and transpose tiles of data. We calculated 2D and 3D DFTs, which were each decomposed into a 1D DFT along each dimension, using DFT kernels that were generated using Spiral [61].

7.2.1 Strided Array Accesses

As a baseline, we first computed a 1024×1024 double precision 2D DFT using row major data. This computation required strided data accesses during column traversal, which were inefficient because they caused misses in the DRAM row buffer. Although our implementation reduced the number of strided accesses by transferring blocks of 16 row elements at a time, we only achieved 40% of peak throughput due to the inefficient strided column traversal.

7.2.2 Tiled Array Accesses

We continued our evaluation using CoRAM++ tiled data layout support, which allowed efficient array traversal along any dimension. Akin, et al. demonstrated this approach for 2D and 3D DFT calculations on FPGAs [8]. Their calculation uses the same Spiral-generated DFT and streaming permutation engines as used in this paper, but uses a custom data transfer engine dedicated to this particular tiled array traversal, and custom data paths between DRAM controllers, permutation engines, and DFT cores.

In contrast, the CoRAM++ 2D and 3D DFTs used the multi-dimensional array agent, allowing run-time selection of array traversal order and in-memory data layout, and an

automatically generated datapath to memory. This flexibility allowed us to use a single FPGA programming file for each DFT size. We computed double precision 2D and single precision 3D DFTs using 4 kilobyte tiles, half of the DRAM row buffer size on the DE4. We evaluated computation performance starting from both row-major and tiled data layouts. Figure 7.2 shows the performance of the following experimental configurations:

1. **CoRAM++ Row-inline** started with row-major data, converted the data to tiled format inline during the first phase of computation, and restored row-major format during the last phase of computation.

2. **CoRAM++ Row-extrapass** was similar to CoRAM++ Row-inline, but restored row-major format using an extra pass through memory after the computation was complete.

3. **CoRAM++ Tiled** started with and kept data in a tiled data layout throughout the computation.

4. **Reference** shows results from [8], which also used a tiled data layout throughout the computation.

Figure 7.2 shows the performance achieved using the various array traversal methods. The **CoRAM++ Row-inline** computation was very slow, because the converting data back to row-major format during the final phase of the computation required the same strided accesses that were required within the row-major computation discussed above. The **CoRAM++ Row-extrapass** computation improved performance in spite of the extra pass through memory because it avoided strided memory accesses. Finally, the **CoRAM++ tiled** computation allowed the CoRAM++ DFT match the performance of the reference calculation and achieve 90% of peak performance.

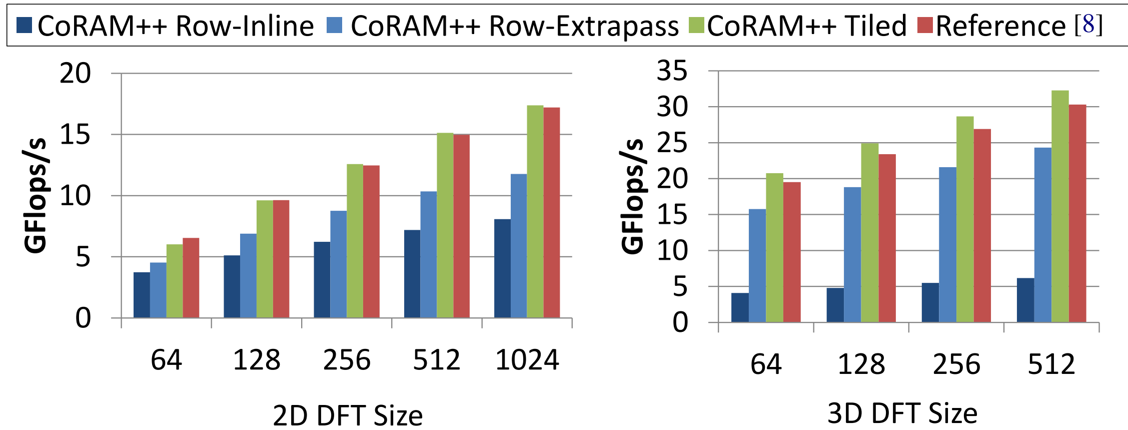


Figure 7.2: 2D (left) and 3D (right) DFT performance.

Table 7.2: Resource utilization for CoRAM++ and reference DFTs.

DFT Type	Logic %	Block Mem %	DSP %
CoRAM++ 1024×1024 2D	62	58	35
CoRAM++ 512×512×512 3D	52	51	20
Reference [8] 1024×1024 2D	27	47	35

Table 7.2 shows that the CoRAM++ DFT incurred higher resource utilization than the reference DFT. While the CoRAM++ multi-dimensional DFT implementation did have more overhead, this overhead paid for the flexibility and ease of use provided by the CoRAM++ programming environment. The reference DFT used a hand-designed data path, memory address generation limited to the tiled array traversal in a fixed dimension order, and did not include support for transferring data to the FPGA from a host computer. In contrast, the CoRAM++ multi-dimensional DFT used an automatically generated NoC for data distribution, supported run-time selection of array layout in DRAM and array traversal method, and included the host computer interface to transfer data between the host computer and FPGA.

7.3 Pointer-Chasing Linked List Accesses

This application scenario shows how data-structure-specific application-level interfaces can simultaneously simplify application development and improve performance. In particular, we show that connecting data-structure-specific components directly to a DRAM interface can portably improve the performance of pointer-based data structures by targeting FPGAs from both Altera and Xilinx, and we show how software accelerators can be used on tightly coupled CPU-FPGA systems to further improve traversal performance while streaming data stored within a linked list to the FPGA fabric.

7.3.1 Linked List Traversal and Merging in Soft Logic

This application scenario traversed linked lists and merged sorted linked lists in place, demonstrating the performance of the hardware linked list agent with and without an accelerator agent. We ran the control threads, kernel, and agents in these experiemnts at 100 MHz. Linked list nodes contained a 4 byte pointer to a data payload and a 4 byte pointer to the next node. Data payloads matched the width of the DRAM controller on each FPGA—64 bytes on the DE4 and ML605, and 128 bytes on the ZC706. We evaluated 5 different ways to traverse and merge linked lists:

1. **Scratchpad-1** is the baseline implementation, and uses a C-language linked list within the application control thread using a typical software construction of linked list operations. It accesses DRAM using the scratchpad agent, which was configured to fetch a single row of data (matching the DRAM interface width) at a time.

2. **Scratchpad-8** is the same as Scratchpad-1 but fetches 8 rows of data at a time when loading data from DRAM to exploit locality and amortize the cost of DRAM ac-

cesses.

3. **Scratchpad-64** is the same as Scratchpad-1 but fetches 64 rows at a time.

4. **Agent-Only** uses the linked list agent to perform linked list operations, and instantiates the hardware linked list engine within the linked list agent (on the same side of the NoC as the application components). The control thread triggers linked list operations, and waits for their completion through function calls provided by the linked list agent. These experiments show the best possible performance achievable using the original CoRAM programming environment, which could not attach data-structure-specific logic to a DRAM interface.

5. **Agent+Accelerator** uses the linked list agent with the hardware linked list engine attached to the DRAM interface as an accelerator. The control thread triggers linked list operations, and waits for their completion through function calls provided by the linked list agent.

7.3.1.1 Linked List Traversal

Linked list traversal experiments were performed on "Best Case" and "Worst Case" DRAM data layouts. The "Best Case" linked list packed linked list nodes and data into contiguous blocks, and the "Worst case" linked list separated linked list nodes and data by 8 kilobytes, causing each DRAM access to miss in the DE4's DRAM row buffer, as shown in Figure 7.3.

Figure 7.4 shows the results of the traversal experiments. In the **Scratchpad** experiments, increasing the number of rows that were loaded by the scratchpad improved performance in the "Best case," but reduced performance on the "Worst case." The reason

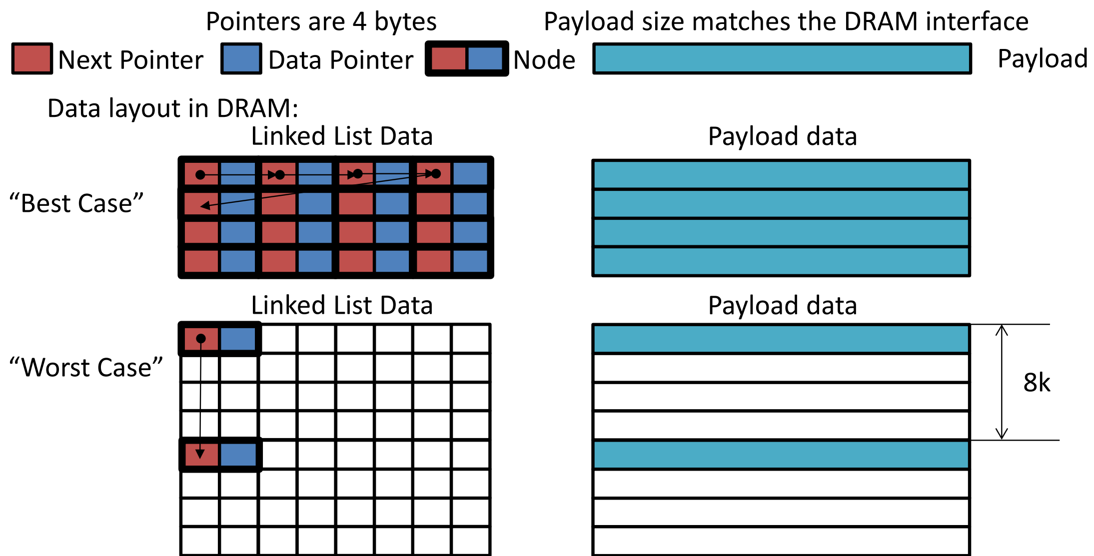


Figure 7.3: Linked list configurations

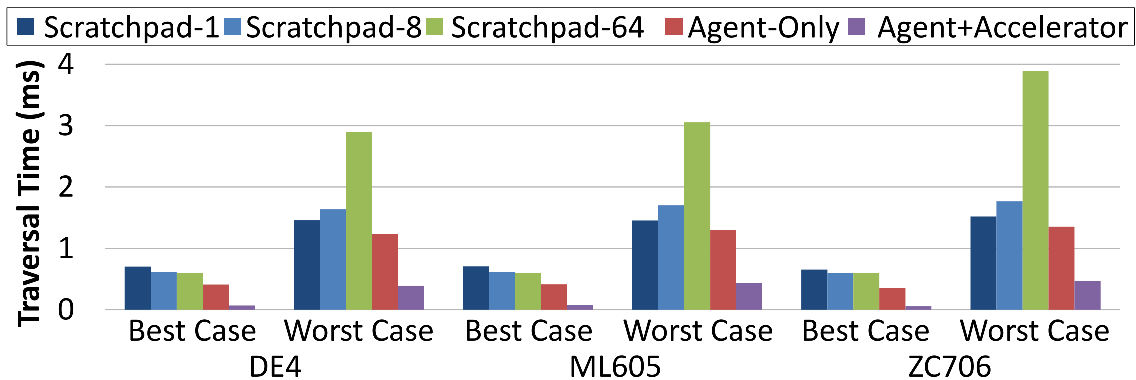


Figure 7.4: Linked list traversal performance

for this was that in the “Best Case,” all of the extra data rows loaded by the scratchpad were used, but none of these extra rows were used in the “Worst Case.” The **Agent-Only** configuration delivered 1.7× the performance of the baseline on the “Best case” data, and was slightly faster than the baseline on the “Worst case” data, due to more efficient linked list data processing.

The **Agent+Accelerator** (with a hardware linked list engine directly connected to a

DRAM interface) provided significant performance improvements, traversing the linked list up to 9× faster than the baseline traversal on the DE4, 8.5× faster on the ML605, and 10.5× faster on the ZC706. This configuration was also up to 5.2× faster than the **Agent-only** configuration. These experiments show that the increased ease of use provided by CoRAM++ data-structure-specific application-level interfaces can also portably improve performance through including data-structure-specific logic attached directly to the DRAM interface.

7.3.1.2 Sorted Linked List Merge

These experiments merged 100 pairs of sorted linked lists which were each created by:

- Randomly assigning each of 100 keys to one list of each pair. The keys were located in a contiguous block of DRAM and aligned to the DRAM interface width of the FPGA.
- Creating a linked list node (pointing to the key) that had a 50% chance of directly following the previous node in the list, and a 50% chance of a random (8-byte aligned) location within a 32 kilobyte address space. This construction simulates linked lists that are modified over time.

Figure 7.5 shows the average run time for each sorted linked list merge implementation, each of which had a standard deviation of approximately 1% of the corresponding average run time. The **Agent+Accelerator** linked list configuration was once again much faster than all other linked list configurations, due to lower latency DRAM accesses when chasing pointers.

Our linked list traversal and merge results show the advantages of accelerators connected to a DRAM interface (reducing the number of NoC round trips), and the general

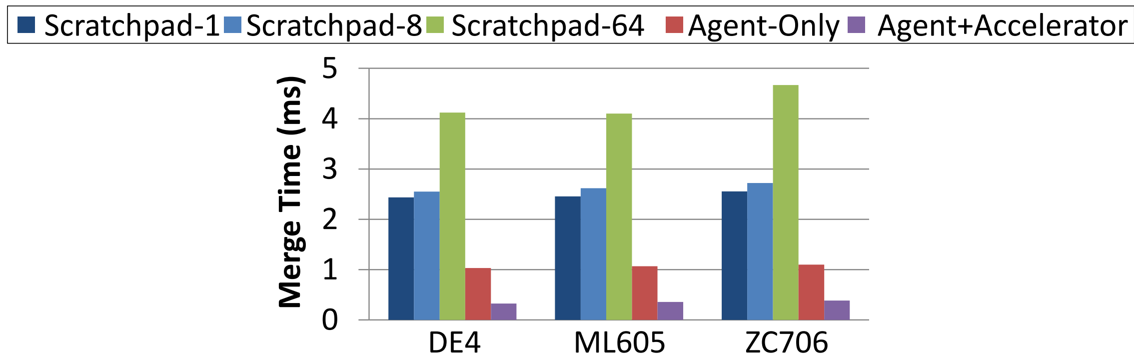


Figure 7.5: Sorted linked list merge performance

advantages of a data-structure-specific approach to managing data transfers. While the exact performance improvement over the original CoRAM is specific to our implementation, any FPGA programming environment that does not allow data-structure-specific logic connected directly to DRAM interfaces would suffer when performing pointer chasing memory accesses.

7.3.2 Linked List Traversal using an ARM Processor Core

Sections 7.3.1.1 and 7.3.1.2 above discuss the advantages of connecting pointer-chasing logic directly to the DRAM interface. This section explores a different mechanism for accelerating pointer-chasing logic. Some of the CPU-FPGA systems (such as the Zynq FPGA used above) that are currently available are tightly coupled, in that the CPU and reconfigurable fabric share DRAM, and may even allow the reconfigurable fabric to access the hard-logic caches attached to the processor cores. There are two reasons why it may be attractive to use a CPU core to accelerate pointer chasing data accesses when using these systems:

1. The hard logic CPU core and attached caches run at much higher clock speeds.

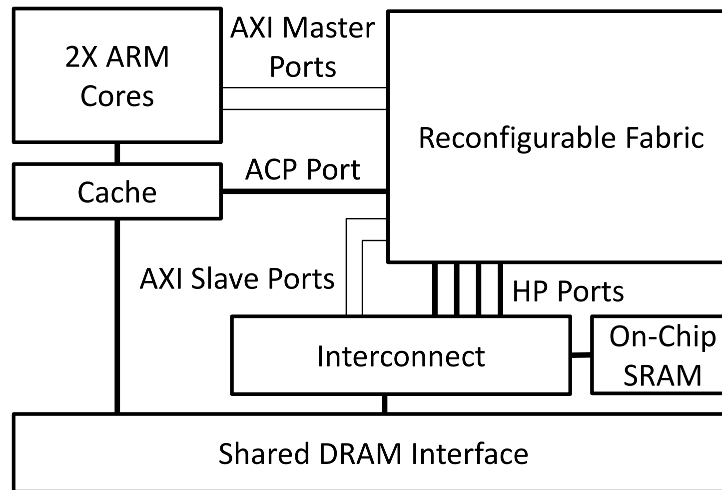


Figure 7.6: Datapaths between memory, the ARM cores, and the fabric in the Zynq

2. Until high level synthesis tools become much better, it is much easier to write software than to create hardware.

Figure 7.6 shows the collection of interfaces available in the Xilinx Zynq. The Zynq includes a reconfigurable fabric paired with two hard-logic ARM A9 processor cores, as well as hard-logic caches, a shared memory interface that maps DRAM and on-chip SRAM (called OCM) into the same address space, and a built-in DMA engine that can transfer data between any pair addresses in this shared address space. The Zynq includes 2 ARM cores running at 667 MHz, an FPGA fabric, a shared DRAM interface running at 533 MHz and double pumped to provide 4.3 GB/s of DRAM bandwidth, and a fabric-only DRAM interface running at 533 MHz and double pumped to provide 12.8 GB/s of DRAM bandwidth, and a 256 kilobyte shared on-chip SRAM that will be referred to as "OCM" in the remainder of this section.

While the experiments in Sections 7.3.1.1 and 7.3.1.2 use the fabric-only DRAM interface to explore FPGA acceleration of pointer-based memory accesses, these experiments use a shared memory consisting of DRAM and OCM, which are mapped into a single address space accessible by both the ARM cores and reconfigurable fabric, and includes the following interfaces to the fabric:

- Four 64-bit wide "High Performance" (HP) AXI slave ports that connect to memory through a dedicated interconnect.
- A 64-bit wide cache coherent "Acccelerator Coherency" (ACP) AXI slave port which accesses memory through the ARM cores' shared L2 cache (and can peek into each core's L1 cache).
- A pair of 32-bit wide general purpose AXI master ports that the ARM cores can use to send requests to the fabric. These ports are each mapped into the address space of the ARM cores, allowing application code to interact with the fabric by writing to or reading to a pre-defined address.
- Two 32-bit general purpose AXI slave ports, which allow the fabric to access memory through several layers of on-chip interconnect.

Each of these interfaces exhibits different bandwidth and latency, which may depend on the memory accesses performed by an application. Applications may choose to balance data accesses across the different interface, in addition to spreading data structure across the ARM cores and processors. For example, a hardware traversal engine instantiated within the fabric can use any of the AXI slave ports to access memory, or a combination of several ports, and can send multiple data requests across any of the ports to interleave memory accesses.

A software implementation could send payload data to the fabric through the AXI master port, but could also buffer data in DRAM or on-chip SRAM and direct a built-in DMA engine to transfer data to the fabric across the AXI port. A hybrid implementation that includes both logic and software could also send pointers to the data payloads to a simple data transfer engine instantiated within the fabric, which could then access the data payloads through any of the memory ports on the chip.

These experiments store linked lists in shared DRAM, and define linked lists nodes as a 4-byte data pointer (stored in-line in contrast to the linked lists above), and a 4-byte next pointer. Our experiments operate on best-case and worst-case linked list data layouts similar to those defined above:

1. The sequential linked list contains 16,384 linked list nodes packed together sequentially as in Figure 7.3 above (except for the in-line data payload), similar to the "best case" linked lists discussed in Section 7.3.1.
2. The strided linked list contains 16,384 each separated by 16 kilobytes, to ensure that every memory requests accesses a different page in DRAM, similar to the "worst case" linked lists discussed in Section 7.3.1. During linked list traversal, we do not attempt to guess that data accesses might be strided and prefetch accordingly, but use this case to ensure inefficient row buffer utilization.

These experiments run the linked list engine at 200 MHz, and attach it to both an HP and ACP port. This engine is controlled by a logic unit that also runs at 200 MHz and accepts commands through an 32-bit wide AXI port, which can be used to direct the the hardware linked list engine, accept streamed data, or access hardware cycle and stall counters.

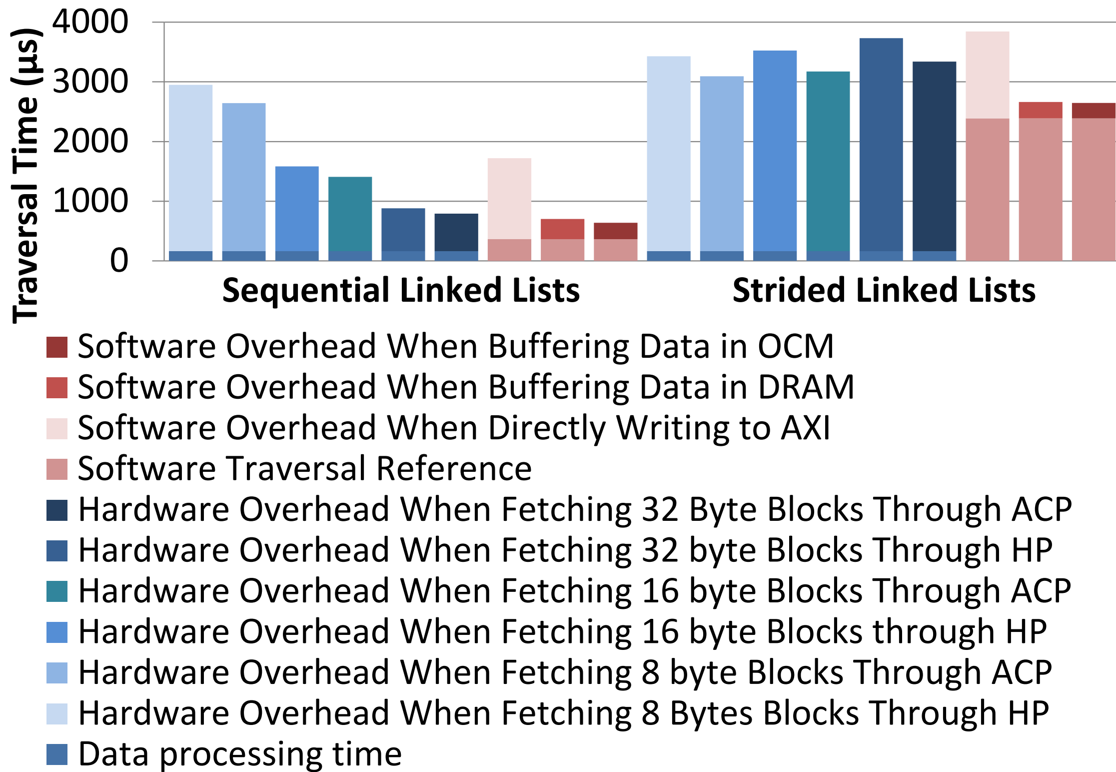


Figure 7.7: Performance achieved traversing linked lists with 4 byte payloads inlined within the linked list nodes.

7.3.2.1 Small-payload lists

Our first experiments investigate linked lists with 4 bytes of inlined payload data per node, which is the simplest case for linked lists.

Summary: Figure 7.7 shows the results of traversing sequential and strided linked lists. We tested the hardware traversal engine using one of the four HP interfaces and using the ACP interface. Our experiments varied the amount of data fetched by the hardware engine to show the effect of retrieving multiple 8-byte words of data (the word width is determined

by the interface width). Experimental results using the hardware engine break the traversal time down into time spent waiting for data and time spent processing it, and experimental results using software traversal break out the time taken when the ARM cores traverse the linked list and touch all of the payload data but do not transfer data to the fabric.

Figure 7.7 also shows software traversal results, either directly writing data to the memory mapped AXI port, or buffering blocks of payload data in either on-chip SRAM (OCM) or DRAM and using the built-in DMA engine to transfer the payload data to the fabric. Since the linked-list payload data starts in DRAM, it is possible to use the built-in DMA engine to transfer payloads data from DRAM rather than aggregating payloads in a buffer, but the performance of doing so was approximately 30× slower than next slowest traversal mechanism.

When traversing the linked-lists using the hardware engine, it is faster to use the ACP port than the HP port. This result holds even when traversing the strided linked-lists, which would prevent a prefetcher in the cache from being effective. Fetching larger blocks of data improves performance dramatically when traversing the sequential linked-lists, because doing so effectively prefetches linked-list nodes. Fetching larger amounts of data reduces performance (by up to about 9%) when traversing the strided linked-lists because the extra cycles spent fetching additional data are wasted. We configure the hardware engine to transfer 32-byte blocks of data for the remainder of our experiments on the Zynq, because the performance improvement of larger data transfers in the sequential case is much greater than the performance degradation in the strided case.

The software traversal experiments also yield interesting results. When the ARM core directly writes payload data to the memory-mapped AXI port, it spends more time writing

data to the port than it does traversing the linked-lists. This is because the Zynq system stalls the ARM core until it has received a response to the bus transaction, which took approximately 100 ns according to our measurements. However, when buffering data and using the built-in DMA engine to transfer data across the AXI port, we find that the DMA engine is able to overlap bus transactions, resulting in much better performance.

We find that buffering data in OCM is slightly faster than buffering data in DRAM. We expected the performance difference to be larger, but believe that the data is primarily being transferred from the cache rather than OCM or DRAM, based upon microbenchmarks that flushed the cache and buffered data without transferring it to the fabric. Figure 7.7 does not include a hybrid traversal, but we have data showing that a hybrid traversal with data payloads this small will not be faster than the software traversal.

Recommendations. Overall, when transferring 4 bytes payloads to the fabric, it is fastest to perform a software traversal, buffer data in OCM, and transfer it to the fabic using the built-in DMA engine. This result is encouraging because the simplicity of writing software code (relative to the effort involved in creating a hardware engine) makes using software traversal desirable.

7.3.2.2 Larger Data Payloads

These experiments investigate the performance of traversing linked-list with larger indirect data payloads. Our hardware traversal experiments use the same hardware traversal engine as in Section 7.3.2.1 above, but use the "payload" provided by the hardware engine as a pointer to the actual payload data, which is used to fetch a block of data with run-time configurable size.

Based the results our earlier experiments, these hardware traversal experiments all

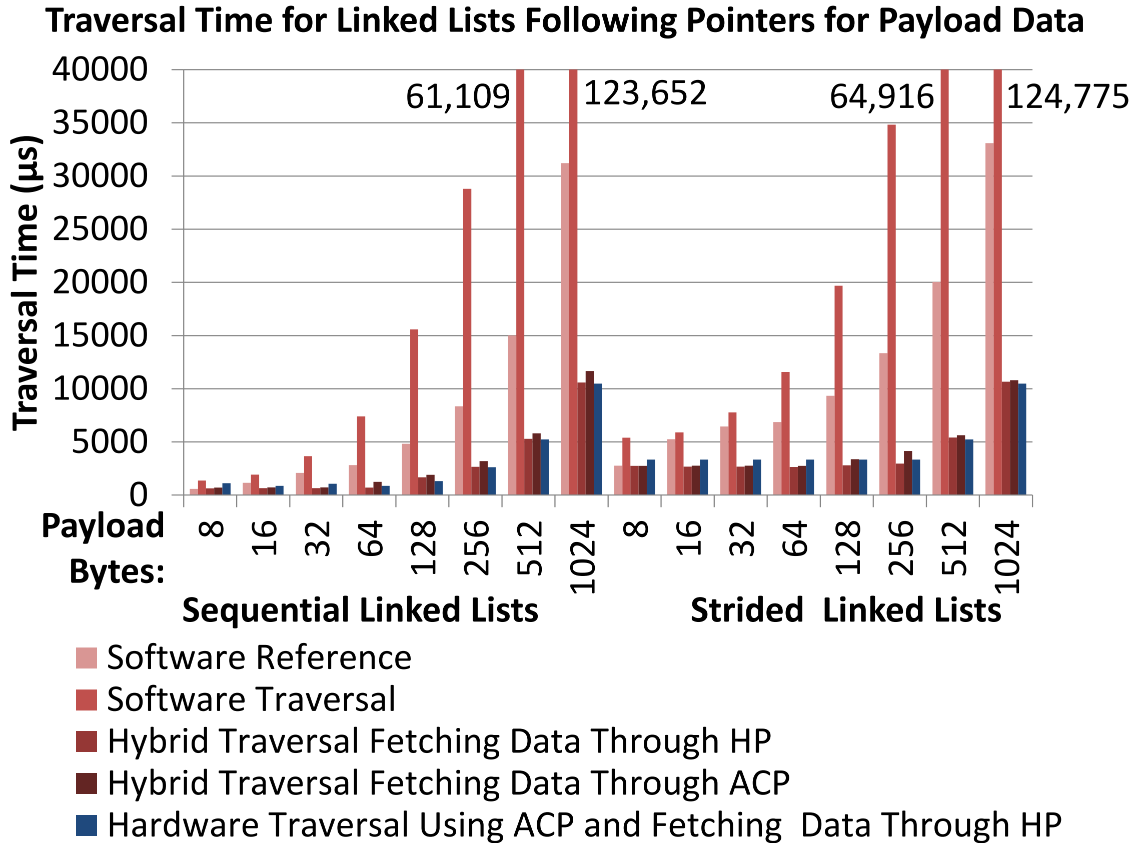


Figure 7.8: Performance achieved traversing linked-lists containing pointers to larger data payloads.

configure the hardware engine to process 32 bytes of data at a time. These experiments either use the ACP or HP port to fetch linked-list data from either sequential or strided linked-lists, and similarly use either the HP or ACP port to fetch data payloads (which are sequential if linked-lists are sequential or strided if linked-lists are strided), resulting in 8 total experiments for each data payload size.

Our software traversal experiments either buffer payload data (in OCM or DRAM) and transfer the data to the fabric using the DMA engine. Similarly, or hybrid traversal

experiments use this mechanism to transfer payload pointers to the fabric, which fetches the payload data using either the HP or ACP interface.The payload size and data fetching interface are configurable at run time.

Summary. Figure 7.8 includes the most interesting results of traversing sequential and strided linked-lists with pointers to payloads varying in size from 8 to 1024 bytes. The figure includes a software reference that touches all payload data but does not send payload data to the fabric in addition to the experiments described above.

The experiments show that accessing larger amounts of data takes longer, as expected, but there are some interesting trends in the data. As the data payloads get larger, the performance differences when traversing sequential and strided lists get smaller. Both the hardware and hybrid traversal mechanisms reach approximately 98% of the peak bandwidth of a 64-bit wide HP interface running at 200 MHz, indicating that the overhead of chasing the pointers (through either mechanism) does not impact performance as much as the time required to transfer 16 megabytes of data. These experiments are also faster than the reference results (which touch all data from the ARM core), indicating that the fabric can achieve higher bandwidth than the ARM core.

The software mechanism is the slowest when transfering more than 4 bytes of payload data due to the lower bandwidth of the master AXI port. The hybrid traversal mechanism is in general slightly faster than the hardware traversal mechanism, and it is slightly faster to use the HP port than the ACP port to fetch payload data, either due to higher streaming bandwidth or reduced contention in the cache. Some of the performance difference of the hybrid mechanism over the hardware mechanism may be due to the fact that we transfer a block of pointers at a time, and fetch data payloads using interleaved requests over the HP

port. Section 7.3.2.3 below shows the performance achieved using interleaved memory requests with the hardware traversal engine.

Additional Large-Payload Results. We also measured the performance that was through larger inlined data payloads,, yielding results similar to those shown in Figure 7.8—with more than 4 bytes of payload data, it is faster to treat the linked list as an indirect payload linked list and use a hybrid mechanism due to the limited bandwidth of the AXI master port. When transferring 4 bytes of indirect payload data, a software and hybrid traversal were about the same speed.

Figure 7.8 excludes a number of experimental results showing similar trends to those exhibited in Figure 7.7. A hardware traversal fetching linked-list blocks that were smaller than 32 bytes were much slower on sequential linked-lists, and slightly faster on strided linked-lists.

When using a hardware traversal, it is slightly faster to traverse the linked-list through the ACP port and fetch data through the HP port than the reverse, and it is a bit faster to spread the data transfers across two different ports than to use the same port for both types of data transfers.

Recommendations. For data payloads larger than 4 bytes, it is fastest to use either a hybrid traversal and stream data payloads over an HP port, but a hardware traversal is only slightly slower.

7.3.2.3 Interleaving Memory Accesses

We've shown that a hybrid traversal is faster than a hardware traversal when payload sizes exceeded 4 bytes, in part because the hybrid mechanism sent payload pointers to the

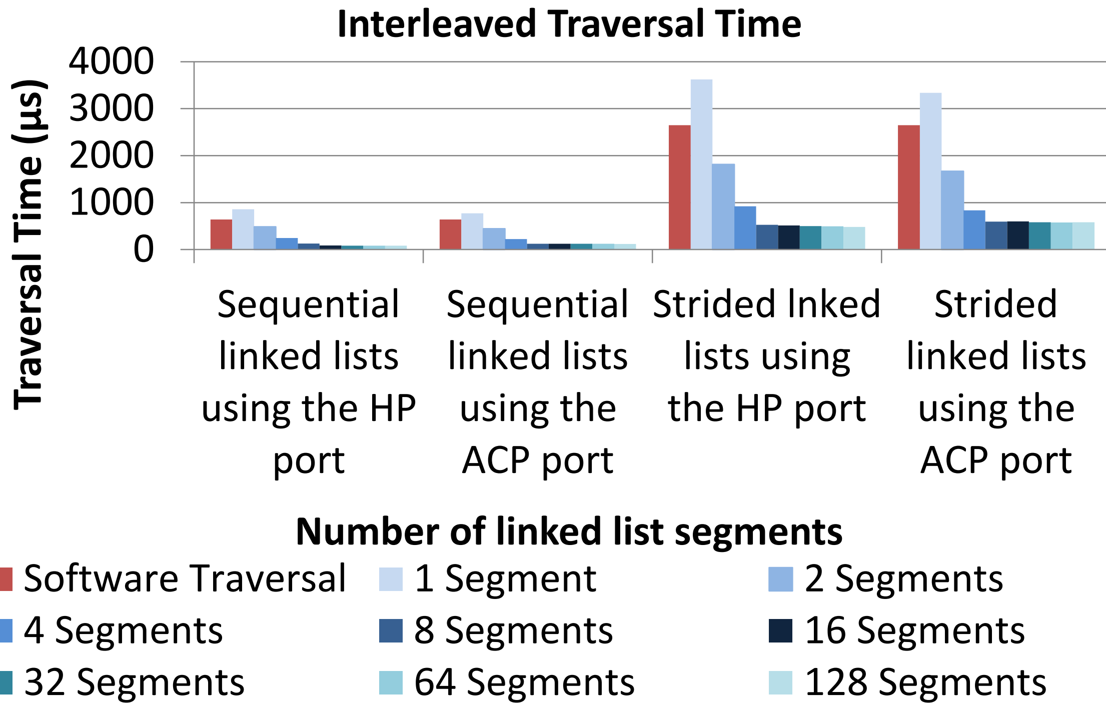


Figure 7.9: Performance achieved through interleaved linked-list traversal. The data points labeled "Software Traversal" are the fastest results shown in Figure 7.3.2.1.

fabric in batches, and interleaved requests for payload data. We've also shown that a software traversal was faster than a hardware traversal with 4-byte payloads. In this section, we investigate whether or not it is possible for the hardware traversal engine to surpass the performance the other two mechanisms by breaking each linked-list into segments and traversing these segments in parallel, similar to a two-layer skip-list [73]. This technique could be used by real-world applications by allowing parallel search, improving the performance of applications that need to access all elements of the list, or extending to parallel traversal of data structures (such as graphs) with multiple pointers per node.

Each experiment in this section breaks a linked-list with 4-byte inlined payloads into

segments and uses the hardware traversal engine to traverse these segments through interleaved requests. These segments simply divide the original linked-lists into equal-sized chunks. We did not attempt to interleave software traversal because the Zynq contains only two ARM cores that do not support hardware multi-threading, limiting the potential benefits of a interleaved implementation. We configured the hardware traversal engine to fetch 32 bytes of data at a time. These experiments loosened our data delivery ordering requirement somewhat—each segment was delivered in order, but we did not address ordering between segments.

Summary. Figure 7.9 shows that it is possible for a hardware traversal to perform significantly better than a software or hybrid traversal. The figure shows the performance achieved using a hardware engine to traverse sequential and strided linked-lists that are broken up into segments, using either the HP or ACP interface. We evaluated up to 128 segments, and the include the fastest software traversal results for reference.

As the number of linked-lists segments increases, so does performance of the interleaved hardware linked-list traversal increases dramatically, until the number of linked-list segments reaches 16. At this point the hardware traversal reachs reach 93% of the peak bandwidth achievable using an 8-byte wide HP interface running at 200 MHz. Increasing the number of segments to 128 allows the hardware engine to achieve over 98% of the peak bandwidth achievable with the HP interface. Performance is slower when using the ACP interface, either due to lower achievable bandwidth or thrashing in the cache. Strided linked-lists achieve approximately 10% of peak interface bandwidth. When performing these test with indirect payload data, strided linked-list performance approximately the same as with inlined payload data, but sequential performance was much worse than with

inlined linked lists, probably because of thrashing the DRAM row buffer when fetching payload data. This was not an issue with the inlined linked-lists because even though the inlined linked-list traversal interleaved memory requests, the hardware engine fetched blocks of 32 bytes at a time, which effectively prefetched data in the sequential linked-list case.

Recommendations. The results presented above clearly demonstrate the benefits of allowing multiple interleaved data requests within the memory subsystem. When segmenting a data structure to allow overlapped traversal, using the HP interface may be faster than using the ACP interface because requests that use the HP interface do not need to go through the cache, and multiple requests will not interfere in the cache. However, if parts of the application that are unrelated to traversal modify data immediately before triggering data structure traversal, it may be more efficient to use a cache-coherent interface than to flush the cache. There are a few caveats to this recommendation:

1. This linked-list segmentation preserved ordering within each segment, but not within the linked list as a whole. If overall ordering preservation is required, a different segmentation method or reordering mechanism should be used.
2. This segmentation mechanism maps very well to linked-lists, which can only be traversed serially, meaning that it is possible to provision the depth of internal queues based on the maximum number of active segments to avoid deadlocks. More complex mechanisms would be required to handle structures with graphs that have a larger branching factor, and that require tracking which nodes were visited.

7.3.2.4 Discussion

Our results allow us to make concrete recommendations for the Zynq-based FPGA that we used, extrapolate these recommendations to other shared-memory systems that include processors and FPGAs, and make recommendations to hardware manufacturers about data pathways between the processors and reconfigurable fabric in these systems.

Recommendations for All Zynq FPGAs: Our results support clear recommendations how to maximize pointer-chasing pefromance on Zynq FPGAs.

1. In general, it is best to traverse the data structure on an ARM core, and easier because the traversal mechanism can be written as software instead of hardware.
2. If data payloads are 4 bytes or less, buffer payloads in OCM and use the built-in DMA engine to transfer them to the fabric.
3. If data payloads are larger than 4 bytes, buffer pointers to payloads in OCM, transfer the pointers to the fabric using the built-in DMA engine, and fetch the data using an HP interface unless cache-coherent access is required.
4. The exception to the recommendations above is that if the data structure supports parallel traversal, it will be much faster to create a hardware traversal engine that can pipeline memory requests, and it may be faster to use one or more HP ports rather than the cache-coherent ACP port if the pipelined memory requests will thrash the cache.

Using Both the Shared and Fabric-Only DRAM Interfaces on the Xilinx ZC706: In addition to the shared DRAM interface, the Xilinx ZC706 board that we used in our experiments also contains a second DRAM interface that is not shared—it only connects

to the reconfigurable fabric. This DRAM interface supports much higher bandwidth than the shared DRAM interface, and can be accessed independently. without concerns about thrashing the DRAM row buffer when interleaving requests for linked list nodes and payload data. Section 7.3.2.2 shows that applications are better off transferring pointers to larger data payloads from the ARM core to the fabric rather than transferring the data items themselves, and these pointers could be used to access data in the fabric-only DRAM interface, which might only contain payload data rather than mirroring the data in the shared DRAM interface. If partitioning data between the shared DRAM interface and the fabric-only DRAM interface, this partitioning might be performed manually by the application developer, or might be partitioned automatically [84].

Other Shared-Memory Processor-CPU Systems: Figure 7.8 also provides guidance into how to best implement pointer-based data structures on systems like the Intel HARP [41] that integrate an FPGA and a processor at the system level. The HARP connects DRAM interfaces directly to the processors in the system, and can potentially also connect a DRAM interface to the FPGA, but allows the fabric to access a NUMA-style single cache-coherent address space. While it should be fastest for the FPGA to access data stored in a DRAM interface directly attached to it, the FPGA can also request data in the DRAM interface attached to the processor, which will be provided through the processor's cache. While fetching results using the cache-coherent ACP interface is slightly slower than doing so using the non-cache-coherent HP interface, fetching data through the cache-coherent ACP interface was faster than touching all of the data using the ARM processor, which required loading each data item into a processor register.

The Xeon processor in the Intel HARP is much more capable than the ARM processor in the Zynq, but also would need to load all data into a register that is narrower than the DRAM interface width in order to aggregate data payloads for an FPGA. This would negatively impact performance when the processor access all of the data, possibly negating the benefits of aggregating the data payloads, especially once the data payloads became larger than a 64-byte cache line.

The Convey HC-1 [22] attaches an FPGA computing system (that includes 4 user-accessible FPGAs) to the front-side bus of an Intel Xeon-based system. This system was designed for FPGA applications that require pointer accesses. The system allows coherent memory access by the processor and reconfigurable fabric, and also attaches special "scatter-gather" DRAM modules to the FPGA fabric, providing up to 80 GBytes/s of bandwidth and allowing low latency random access to 8-byte data blocks. This memory system would greatly reduce the performance reduction of the "strided" lists when compared to the "sequential" lists, with the result that in this system it would always be beneficial to traverse pointer-based data structures using a hardware traversal engine.

Recommendations for Future Hardware: Our experimental results also allow us to make recommendations to device manufacturers who are designing future shared-memory hybrid processor-FPGA systems. It does make sense for a reconfigurable fabric to be able to access both a cache-coherent interface optimized for processor-FPGA accesses in addition to bandwidth-focused non-cache coherent interfaces, but it would behoove device manufacturers to provide better guidance as to how to best utilize these disparate interfaces.

Device manufacturers should provide processors with simple memory-mapped access to the reconfigurable fabric, but should allow the processors to easily transfer data to the fabric without blocking (assuming that internal queues are not full), rather than needing to manually buffer data and use a DMA engine—whose management also incurs run-time overhead. Providing better guidance about how to use the various memory interfaces available to the reconfigurable fabric, and allowing a processor to directly transfer information to the fabric without blocking would make it much easier for application developers to create applications that cross the processor and reconfigurable fabric and achieve the best possible performance.

7.4 Graph Computations Using the Gather-Scatter Work-list Agent

The gather-scatter work-list agent was evaluated in the context of the GraphGen compiler [67, 82], which supports computations on data associated with the vertices and edges of a graph, and assumes that the structure of the graph does not change during the computation. The GraphGen compiler operates on a vertex-centric description of graph computations, which is depicted in Figure 7.10. The application developer provides the following information:

1. The structure of the graph.
2. The format of data associated with each vertex and edge.
3. The required graph traversal order.
4. The computation to perform at each vertex, which invokes accelerator kernels included with the graph specification.[1]

[1] The computation operations on data associated with the vertex, adjacent vertices, and the connecting

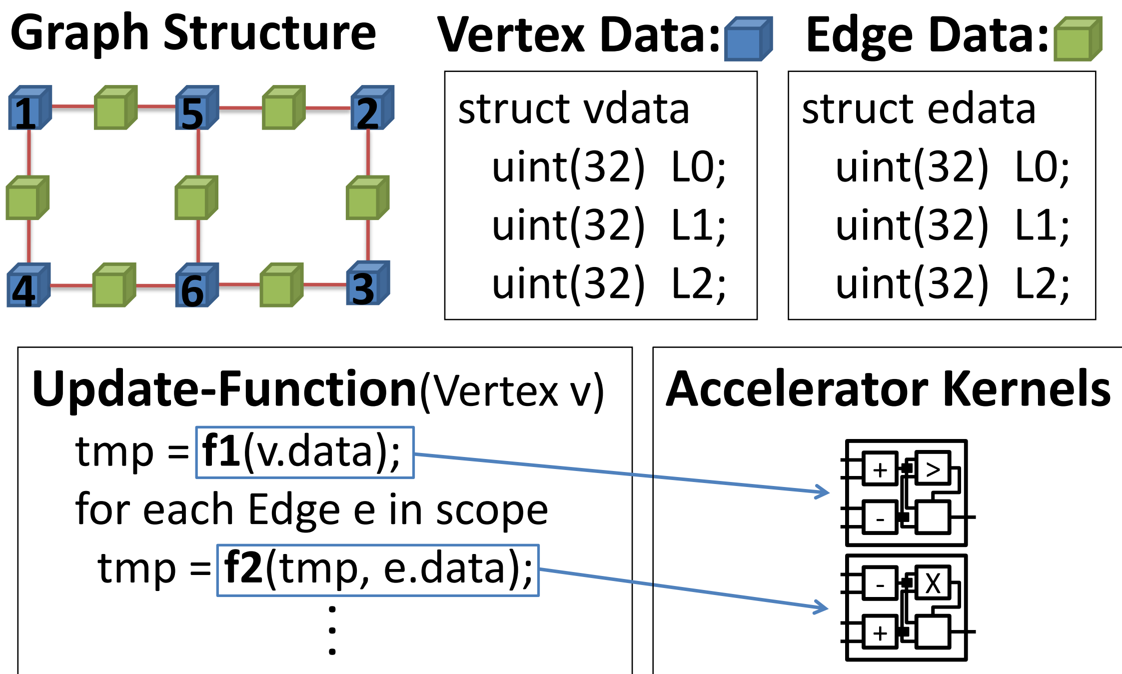


Figure 7.10: Vertex-centric graph description.

The GraphGen compiler supports an instruction-based hardware kernel for each application that invokes the application's accelerator functions with appropriate vertex and edge data as specified by the application designer. This hardware kernel is multi-threaded in that it keeps track of multiple in-flight vertex computations, and supports multi-ported vertex and edge memories that are used to partially unroll the loop over each vertex's neighbors.

The key observation behind the GraphGen compiler is that if the structure of the graph does not change during a computation, it makes sense to pre-process the graph in order to make data transfers more efficient. Since vertices and edges in the graph may be part of more than one sub-graph, accessing sub-graph elements requires pointer-based memory accesses, making the gather-scatter work-list agent a good fit for this application, which

edges.

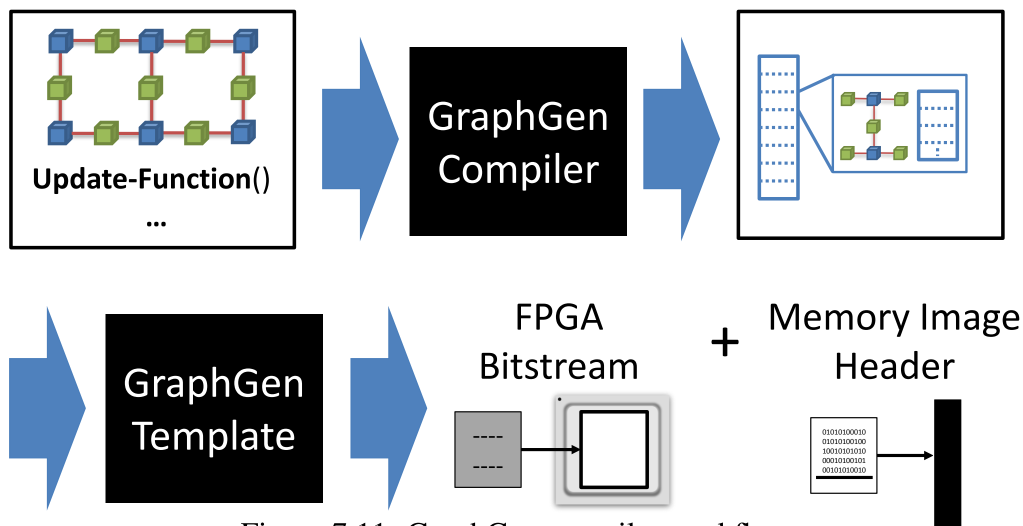
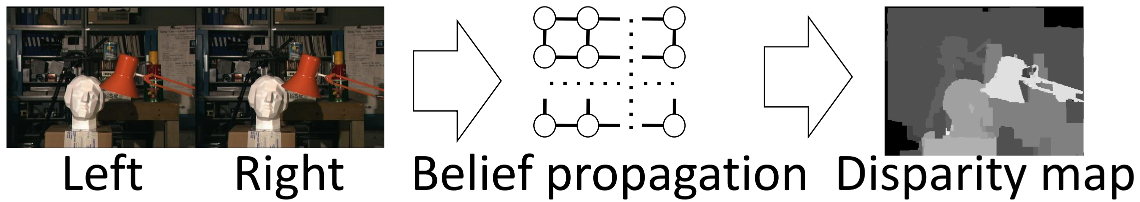


Figure 7.11: GraphGen compiler workflow.

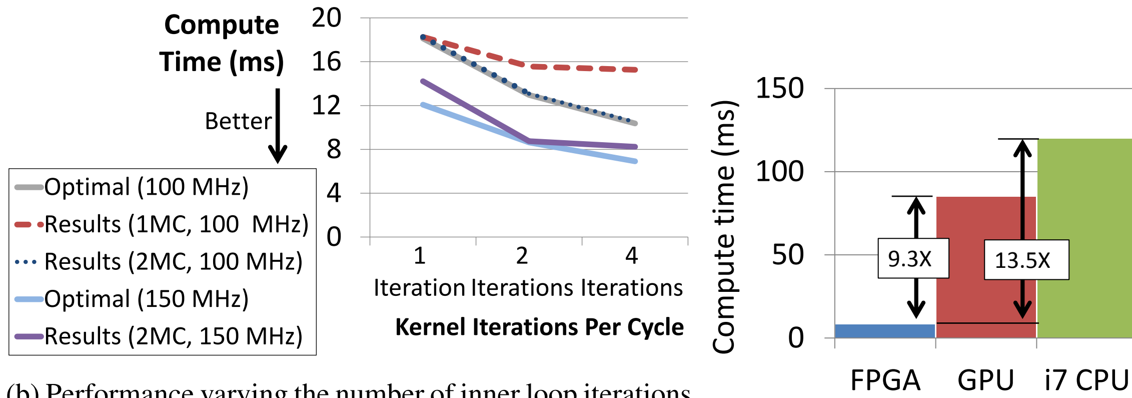
stores vertex data in one buffer and edge data in another buffer within the gather-scatter work-list agent.

The general flow of the GraphGen compiler is shown in Figure 7.11. The user provides the graph structure and an update function to apply to each graph vertex. This graph structure goes through a front end compiler, which breaks the graph computation into a sequence of computations on sub-graphs. Each sub-graph computation is mapped to sub-task for the gather-scatter work-list agent, which applies the optimizations discussed in Section 6.5 to each sub-task.

We evaluated the GraphGen compiler (and gather-scatter work-list agent) on two different graph applications: depth reconstruction using Tree ReWeighted Stereo Matching (TRW-S, shown in Figure 7.12a) and handwriting recognition using a Convolutional Neural Network (CNN, shown in Figure 7.13a). We targeted both Xilinx and Altera FPGAs, and used both interfaces on the Altera FPGA in order to achieve higher DRAM bandwidth.



(a) Depth Reconstruction using Tree ReWeighted Message Passing. A pair of images from a stereo camera is converted to a grid graph, and the algorithm recovers the depth at each pixel in the grid graph.



(b) Performance varying the number of inner loop iterations per cycle, number of DRAM interfaces and kernel clock speed.

(c) Performance compared to best effort CPU and GPU implementations

Figure 7.12: Depth Reconstruction application configuration, raw performance and performance comparison

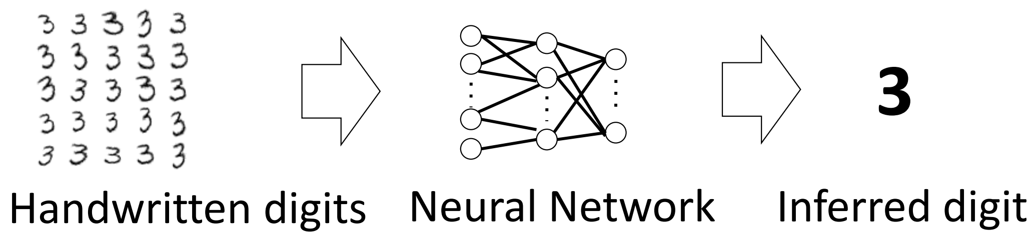
7.4.1 Depth Reconstruction using Tree-ReWeighted Message Passing

Depth reconstruction is the process of inferring the depth of objects in a images from a stereo camera by determining parallax—how much each point moves between the two images. Figure 7.12a shows how each pair of pixels in the image is mapped to a grid graph, which is processed using a belief propagation algorithm called Tree-ReWeighted Message Passing [51]. This algorithm can be mapped efficiently to FPGAs using a diagonal parallelization scheme [27].

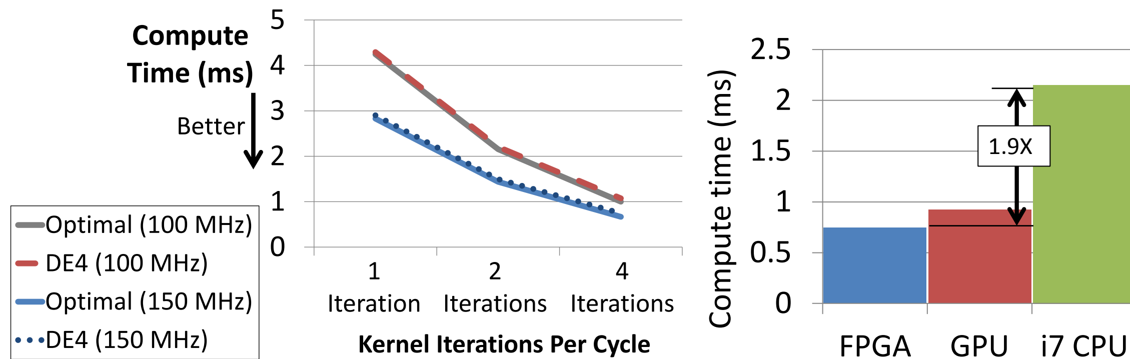
Our experiments explored the effectiveness of unrolling the innermost loop of the computation, increasing the number of read ports in the local buffers appropriately and varying the clock speed of the computation kernel, and seeing how well the gather-scatter work-list agent could support the application the amount of time take by the computation decreased, increasing the bandwidth needed to saturate the computation kernel. We tested this application on both the ML605 and DE4 boards, and used either one or two DRAM interfaces on the DE4 (increasing the data path width when using 2 memory controllers to allow full-bandwidth data transfers).

Figure 7.12b shows the results of these experiments, operating on the "Tsukuba" images of the Middlebury benchmark[75]. The ML605 and DE4 boards performed identically when the DE4 was configured to use a single memory interface. The gray and blue solid lines show "optimal" performance, as defined by the amount of time that the computation could take given the clock speed of the kernel and the number of cycles needed by the computation. The dashed red line shows that with 1 memory controller and a kernel running at 100 MHz, the gather-scatter work-list agent can saturate the kernel when the computation's inner loop is not unrolled at all, but is bandwidth limited when the computation's inner loop to compute 2 or 4 inner loop iterations per cycle. With 2 memory controllers, the gather-scatter work-list agent can always saturate the a kernel running at 100 MHz. When the kernel's clock is increased to 150 MHz, the application becomes bandwidth limited after the kernel's inner loop is unrolled once. We achieved 85% of peak DRAM bandwidth during this computation, because some strided data transfers were necessary due to data reuse across subgraphs.

Figure 7.12c shows that performance is quite favorable when compared to best-effort



(a) Handwriting Recognition using a convolutional neural network. The input image is used as the input to the first layer of a neural network that recognizes handwritten characters



(b) Performance varying the number of inner loop iterations per cycle and kernel clock speed.

(c) Performance compared to best effort CPU and GPU implementations.

Figure 7.13: Handwriting Recognition Application Configuration, Raw Performance and Performance Comparison

implementations of the algorithm on a high end Core i7 CPU and Nvidia GTX 680m GPU, achieving 13.5× better performance than the CPU and 9.3× better performance than the GPU. Even though this algorithm operations on a grid graph, the diagonal parallelization of the algorithm does not perform well on a GPU because of the data access patterns that it creates. The applications compiled by the GraphGen compiler needed all available DRAM bandwidth to achieve the best possible performance, so would not benefit from the ShrinkWrap extension to the CoRAM architecture.

7.4.2 Handwriting Recognition Using a Convolutional Neural Network

We also evaluated the gather-scatter work-list agent on handwriting recognition using the the well-known LeCun [56] convolutional neural network. Figure 7.13a shows how the input images are mapped to the input layer of the neural network, which is used to infer a character. We used a publicly available, pre-trained neural network for handwriting recognition [69]. We once again evaluated unrolling the loop and running the kernel at 100 or 150 MHz. using both DRAM interfaces on the DE4. Figure 7.13b shows that the gather-scatter work-list agent can saturate the kernel in all experiments, and Figure 7.13c shows that the FPGA was about 2$\times$ faster than the Core i7 CPU, and slightly faster than the GPU.

Since the publication of the paper [67], the CoRAM backend for the GraphGen compiler has been updated to support the ZedBoard, an FPGA that incorporates a Xilinx Zynq chip. The Xilinx Zynq chip includes an FPGA fabric along with two hard ARM cores, and the CoRAM backend supports transferring data to the board over Ethernet, through a program running on one of the ARM cores that initializes data using a DRAM controller that is shared between the ARM core and FPGA fabric.

The key takeaway from the GraphGen project is that when a graph structure is fixed, it is possible to preprocess the graph to reduce the number of irregular data transfers, and use the large amount of local storage available on FPGAs to achieve better performance than CPUs and GPUs which have much more DRAM bandwidth.

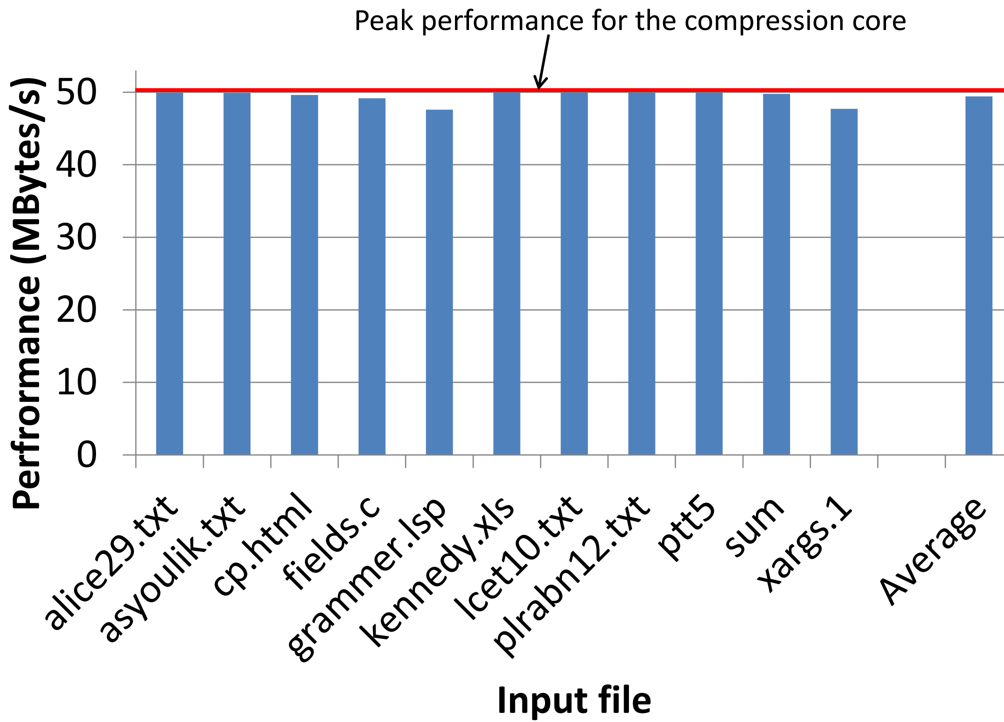


Figure 7.14: Compression kernel performance.

7.5 Streaming Data Compression

This section evaluates a compute-bound application in the form of a publicly available data compression core [4]. The data compression core implements the LZRW1 [83] compression algorithm, which is a variant of the well-known Lempel-Ziv [97] data compression algorithm that is designed to be fast.

We evaluated data compression using the ZedBoard, using a CoRAM++ application that uncompressed streamed data to the compression engine (which ran at 100 MHz), and streamed compressed data back to DRAM, after which it was transferred to the host system for verification.

Figure 7.14 shows the throughput of the compression kernel that was achieved when compressing all of the files in the Canterbury Corpus [18]. The figure also includes the mean performance achieved, and includes a red line showing the peak achievable running this compression core at 100 MHz. The application is very close to the peak achievable performance in all cases. Performance was limited by the compression core, which could only accept one input byte every two cycles.

7.6 Kernel-Kernel Communication

Our evaluation of the kernel-kernel communication agent demonstrates that it can support full-bandwidth data transfers between arbitrary kernels, and quantifies the resource overheads introduced by the extra NoC endpoints that the agent introduces. Our experiments instantiate simple hardware kernels that read data from a read stream agent and immediately write the data to a write stream agent. Each experimental configuration instantiates a different number of hardware kernels, along with an appropriate set of kernel-kernel communication agents to support a chain of data transfers from one of the DE4's DRAM interface, through each of the hardware kernels, and out through the other DRAM interface.

The CoRAM++ NoC was used in a standard configuration that divides the application into DRAM controller (and kernel-kernel communication agent) endpoints and application endpoints, and creates a crossbar between the two sets of NoC endpoints. The NoC was configured for a data width matching that of the hardware kernels and DRAM interfaces on the DE4, which was 512 bits at 100 MHz. For each experimental configuration, we streamed data sets that varied in size from 8 kilobytes to 16 megabytes.

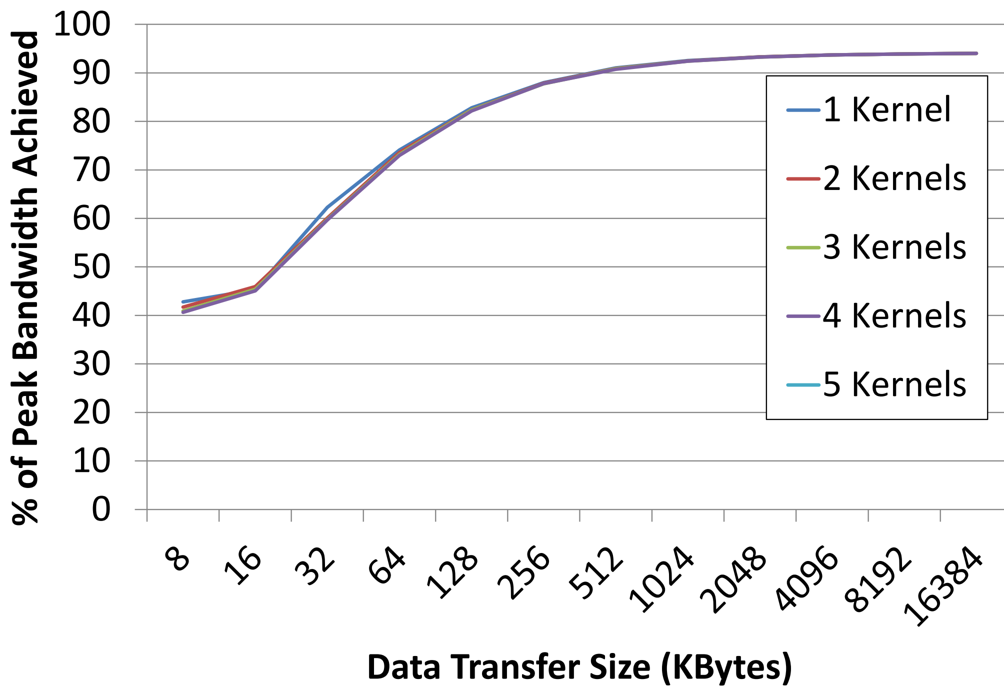


Figure 7.15: Kernel-Kernel Communication Agent Performance Results.

Figure 7.15 shows that applications using the kernel-kernel communication agent to stream data between kernels does not hurt memory access performance. The baseline result is a 1 application kernel configuration that simply streams data in from DRAM and out to DRAM. This kernel achieves the same performance as the streaming DFT discussed in Section 7.1, except for the fact that performance on small data transfers is not hampered by the latency of the DFT kernel. As the number of kernels (and trips across the NoC) increases, the bandwidth achieved by the application stays the same, indicating that the kernel-kernel communication agent can efficiently transfer data between application kernels.

Figure 7.16 shows how resource utilization changes as the application is rebuilt with

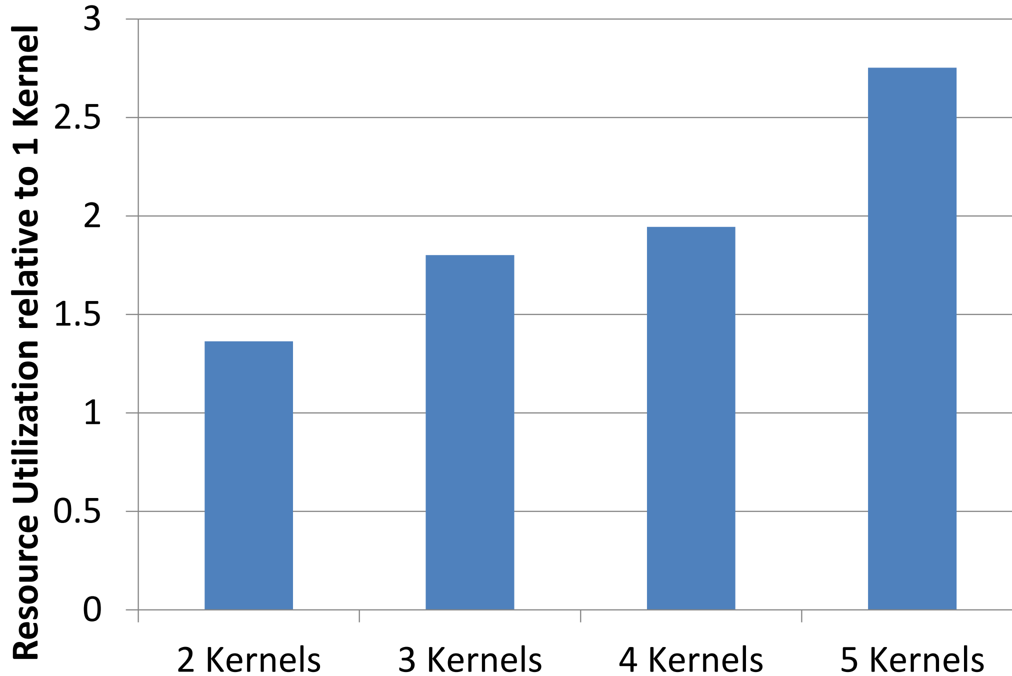


Figure 7.16: Kernel-Kernel Communication Application Resource Utilization.

an increasing number of hardware kernels and kernel-kernel communication agents. The figure shows resource utilization relative to the "1 Kernel" configuration, based on the "Fitter resource utilization summary" reported by Quartus. Increases in resource utilization are due to the additional kernels, additional kernel-kernel communication agents, and larger NoC needed to support them. The "1 Kernel" baseline configuration includes a 2×4 crossbar—2 NoC endpoints on the DRAM interface side of the NoC, and 2 NoC endpoints on the application side. This configuration allows the application to transfer data at the full bandwidth provided by the DRAM interfaces on the DE4. The application included the host communication agent, which was configured as in Figure 6.6b, and is the reason why the application side of the NoC requred 4 NoC endpoints. While it is possible

for CoRAM++ agents to share NoC endpoints, this can reduce the available bandwidth available to each CoRAM++ agent.

Increasing the number of kernels to 2 increased the NoC size to 4×6, as it added 2 NoC endpoints on the DRAM interface side of the NoC for the single kernel-kernel communication agent, and 2 NoC endpoints on the application side of the NoC for the additional stream agents. The size of the application increased by 40%. As the number of application kernels increased, the number of NoC endpoints on each side of the NoC increased by 2. However, resource utilization did not increase linearly due to quantization effects within the FPGA. Quartus also skipped some fitter optimizations in all cases in order to save time, because the application was not big enough to completely fill the FPGA.

Chapter 8

Conclusion

FPGAs are growing more capable every device generation, taking advantages of the same increased transistor counts that Moore's law grants to other computing devices. As FPGAs grow larger and more capable, FPGA applications can afford to expend resources on frameworks that simplify application development and provide application portability. Simplifying FPGA application development will increase the adoption of FPGAs as general purpose FPGA computing devices, helping to satisfy the ever-growing need for power efficient high performance computation.

CoRAM++ proposes that application developers should use application-level memory interfaces that are specific to the data structures in use. CoRAM++ supports these types of interfaces through an FPGA programming environment with a two-layer system architecture: a low-level system interface provides communication services, and an extensible library layer provide data-structure-specific application-level interfaces. An essential part of these interfaces is that the application-level abstraction makes the application oblivi-

ous to the implementation interfaces, which can include an optimized datapath to memory that directly connects pointer-chasing logic to a memory interface, or even runs software pointer-chasing code on a hard-logic processor. These interfaces are the key to providing convenience to the application developer without adding too much overhead in terms of run-time performance or resource utilization, which should make CoRAM++ application developers.

Future Work

This thesis has shown the effectiveness of CoRAM++ for supporting some important data structures and applications. There are several ways in which CoRAM++ can be expanded to provide more utility to FPGA application developers:

- **More CoRAM++ targets:** CoRAM++ currently supports a small subset of currently available FPGA boards. This is enough to demonstrate the portability benefits of CoRAM++, but support should be extended to additional hardware platforms. There is no reason why CoRAM++ needs to directly target FPGA boards—an effort is currently underway to use LEAP as a back-end target for CoRAM++, instantly conferring support to all of the FPGA boards that LEAP can target.
- **More data structures:** The CoRAM++ library currently supports some of the data structures that are important to FPGA computing, but could support more of them. In particular, graphs are very popular in the machine learning domains, and better, accelerated support for computations on graphs could allow CoRAM++ to appeal to a wider audience
- **A deeper investigation of software pointer-chasing acceleration:** Section 7.3.2

demonstrated an initial evaluation comparing the traversal of pointer-based data structures in both hardware and software on tightly coupled CPU-FPGA systems. This study can be extended to more data structures and hardware platforms, including Intel's HARP, which couples state-of-the-art Xeon processors with high end Altera FPGAs.

- **More applications:** There are many applications that could be used to further demonstrate the benefits of CoRAM++. In particular, the multi-dimensional DFT used demonstrated in Section 7.2 could be used in conjunction with the kernel-kernel transfer interface within a Synthetic Aperture Radar (SAR) application.
- **Underlying implementation:** CoRAM++ heavily relies on the Bluespec System Verilog (BSV) language, which is convenient, but expensive and not open source, making the barrier to entry to using it high. This infrastructure could be replaced by an open source language such as Chisel [20] that supports similar functionality.

Appendix A

Application Notes for Supported Compiler Targets

A.1 General Application Notes

As the CoRAM++ abstraction is meant to help develop applications on current and near-future FPGAs, most of the experiments discussed in this chapter were performed on FPGAs rather than in simulation. The streaming (Section 7.1) and multi-dimensional array (Section 7.2) experiments were performed exclusively on the DE4, as they read data from one DRAM controller, through the application, and write it to the other DRAM controller as quickly as possible. These experiments would achieve poor performance on the ML605 or Zynq-based FPGAs due to contention between read and write accesses.

On the other hand, the experiments in Section 7.3 were performed on the ML605, DE4, and ZC706 boards, and those in Section 7.4.1 utilized the ML605, and DE4, showing the portability of CoRAM++ applications. No application code was changed when moving

these applications between boards, as these application used vendor-agnostic wrappers for block memories, floating point cores, and other vendor-specific structures that are provided by the CoRAM++ compiler.

Except where otherwise noted, all experiments used the Host Communication Agent to initialize data in FPGA DRAM, count the number of clock cycles that application execution took, and read data from FPGA DRAM to the host computer for validation.

A.2 ML605 Compiler Target Notes

The Xilinx ML605 is a Xilinx Virtex-6 based FPGA board containing a single DDR-3 DRAM controller connected to 512 megabytes of DRAM. The Xilinx Memory Interface Generator was used to create the top level module for this board, which includes the DRAM controller and clock generators. The DRAM controller provides a DRAM interface that is 32-bytes wide and runs at 200 MHz, providing 6.4 gigbytes/second of bandwidth. The DRAM controller allows accesses to data at 32-byte aligned address. and supports a 27-bit address that is measured in 8-byte words.

The MIG-derived clock generation logic was modified to generate 50, 100, and 150 MHz clocks, which are provided to the application—CoRAM++ compiler flags can be used to pass these through to the application kernels. The 100 MHz clock is also used for the NoC, Control Thread state machines, and other infrastructure components except where otherwise indicated. Since infrastructure and application components run at 100 MHz, the CoRAM++ memory subsystem always accesses 64 bytes of data per request, and internally converts two 32 byte DRAM controller requests into one 64 byte request to allow applications to achieve all available DRAM bandwidth. Applications needing all

of the available DRAM bandwidth at all application-side NoC endpoints should specify a 64-byte NoC width.

The ML605 contains a serial UART that is connected to a USB port. Since this serial connection is actually connected through high speed USB wires rather than serial wires, the host computer can communicate with the serial port at 500 kilobits/second. The ML605 documentation implies that its serial connection may actually support connections up to 1 megabit/second, bit this functionality was not tested. On this board, the serial port is generally mapped to address 0x80000000 in a 32 bit global address space.

A.3 DE4 Compiler Target Notes

The Terasic DE4 is an Altera Stratix-IV based FPGA board containing two DDR-2 DRAM controllers each connected to 1 gigabyte of DRAM. Two variants of the FPGA board are available: one with a smaller FPGA containing 228 ,000 logic elements, one with a larger FPGA containing 531,200 logic elements. All experiments were performed with the larger FPGA. The DE4 target for the CoRAM++ compiler uses the modern UNIPHY controller IP rather than the older ALTMEMPHY controller. Like the ML605 board, each DRAM controller communications through a 32-byte wide channel that runs at 200 MHz, and the CoRAM++ compiler target converts this communication channel to a 64-byte wide channel at 100 MHz, and always requests two rows of data at once. Unlike the ML605 compiler target, the DE4 compiler target uses 27-bit addresses that point to 32-byte words (address 1 points to bytes 32-63).

Initial implementations of the DE4 CoRAM++ compiler target used the ALTMEMPHY controller, but we saw data corruption under high load, possibly due to the issue that

Altera discloses in [10]. Since the ALTMEMPHY memory controller cannot be created with newer versions of Quartus, the CoRAM++ target was updated to use the UNIPHY DRAM controller.

Another issue with the DE4's DRAM controller implementation is with the reference clock pin that drives the second DRAM controller. Quartus does not accept the clock pin that Terasic recommends as a valid reference clock pin, and issues "critical warning" messages when using the second memory controller. The Altera Forums contain a short discussion of this issue and a resolution using a different clock pin [11].

The DE4 contains a serial UART that connects to a DB9 serial cable. The CoRAM++ compiler target can communicate with this UART at a maximum of 115,200 kilobits/second, and maps the serial UART to address 0x80000000 in the global address space.

A.4 ZedBoard and ZC706 Compiler Target Notes

The ZedBoard is an inexpensive ($395 retail price, $319 academic pricing as of 02/02/2015 [96]) Zynq-based FPGA board. This board contains two hard ARM M9 cores connected to hard caches, a hard AXI bus, and a small FPGA fabric (See Table 5.1), and can run both Linux and Android [37], making it an interesting platform for exploring both mixed CPU-FPGA designs and mobile applications that use FPGAs.

One downside of the ZedBoard is that even though its DRAM controller runs at 533 MHz (1066 megabits/s/data pin), the chip's DRAM data path is limited to 32 bits (rather than the standard 64 bits), limiting DRAM bandwidth to 4.3 gigbytes/s. The same limitation applies apply to the shared DRAM controller on the larger ZC706 FPGA board [91], but Altera's comparable Arria SoC devices achieve higher throughput through higher clock

speeds in their DRAM controllers [14]. When targeting Zynq platforms, note that there is an error in the way that both ISE and Vivado configure the shared DRAM controller [87], and the System-on-Chip initialization code must be manually updated on the build machine.

The ZedBoard contains four high performance 64-bit AXI ports that connect the FPGA fabric to the DRAM controller, and an additional 64-bit AXI port that is coherent with the ARM cores' processor caches. The CoRAM++ compiler target Zynq and the ZedBoard can aggregate several AXI ports together into one logical interface in order to saturate the onboard DRAM controller of the ZedBoard, and has successfully connected to these AXI ports at 200 MHz. Since the connection is to a generic AXI port, this interface can also connect to other AXI devices, and from the perspective of the CoRAM++ compiler target it does not matter if it is connecting to the cache-coherent AXI port or one of the non cache-coherent AXI ports. These AXI ports accept 32 bit addresses that point to bytes and must be aligned to 8-byte boundaries.

Since the ZedBoard's DRAM controller is shared between the ARM cores and reconfigurable fabric, the reconfigurable fabric must not arbitrarily access all of the FPGA dram, but must used a region of DRAM that is reserved for it by the software running on the processor. The CoRAM++ compiler target for the ZedBoard and ZC706 supports an offset that can be added to every request before the request is passed to the AXI ports. This offset is configured through the Host Communication Interface, but generally configured by the ARM-based program that performs the FPGA side of the interface rather than the actual host system.

On the ZedBoard, clock and reset signals are generated by the ARM processing system.

Clock signals running at 50, 100, 150, and 200 MHz are provided, and control threads are generally configured to run at 50 MHz to make it easier for ISE to fit the application onto the small FPGA fabric. Our experimental results have shown that this low clock speed does not negatively impact application performance.

The CoRAM++ compiler target for the ZedBoard and ZC706 exports the clock signals, reset signals, and AXI ports from the Xilinx Platform Studio (XPS)- based processing system to a top level verilog module that can be used with ISE. When building this compiler target, we found that the Zynq chip did not enable the clock signals, reset signals, and AXI ports unless we included a Xilinx AXI bus IP – that is we needed to include this IP even though the AXI ports are part of the ARM cores. It appears that the ARM initialization code generated by XPS does not activate the clock signals, reset signals, and AXI ports unless the AXI bus IP was included, and this issue also occurred with Vivado. Even though multiple AXI ports are exposed, the AXI bus only needed to be connected to a single AXI port (even one that does not have any application components connected to it) for the the clock signals, reset signals, and AXI ports to be active, which supports the theory that the problem lies with the initialization code.

The CoRAM++ Host Communication Agent connects to a single AXI slave port on the Zedboard and ZC706– the ARM cores send commands to this port, and the Host Communication Agent responds. This AXI port is 32 bits wide, and generally is configured to run at 50 MHz on the ZedBoard, as it is not carrying high-speed information.

Since the ZedBoard contains a very small FPGA and a single DRAM controller, the CoRAM++ compiler can omit the CoRAM++ NoC when targeting it, attaching the CoRAM Clusters that drive the CoRAM Classic SRAM Agents directly to the 64-bit AXI ports dis-

cussed above.

The ZC706 is similar to the ZedBoard, but contains a much larger FPGA fabric, and contains a second DRAM controller, which runs at 800 MHz and provides up to 12.8 gigabytes/s of DRAM bandwidth. This DRAM controller is connected to the CoRAM++ compiler target through a 64-byte interface that runs at 200 MHz, and converted to a 128-byte interface running at 100 MHz to match the clock speed of the NoC and other infrastructure. This interface accepts 28-bit addresses that point to 8-byte words, which must be aligned to multiples of 8 [1] Both DRAM controllers are attached to the CoRAM++ NoC, but the shared DRAM controller is expected to be initialized through the ARM cores.

Since the second DRAM controller is not connected to the ARM cores, the Host Communication Agent must stream data to or from this DRAM controller through the AXI slave port attached to the ARM cores, and consequently runs the slave AXI port (which is still a 32-bit wide port) at 100 MHz. Clock and reset signals are generated by the second DRAM controller when targeting the ZC706. The compiler target provides 50, 100, 150, and 200 MHz clock signals.

The software running on both the ZC706 board's ARM cores to implement communications is the same as that in the ZedBoard, as is the ethernet-based communication protocol (using the Lightweight IP ethernet stack). The ethernet controller is currently configured to use a static IP address of 192.168.1.10 and run at 100 megabits/s. This software currently supports the "standalone" system library.

[1] The first valid address is 0; the second valid address is 8, which will points to bytes 64-127 in the 200 MHz clock domain. The CoRAM++ template for this FPGA requests 128-byte aligned data, so would request address 0 for bytes 0-128, 16 for bytes 128-256, and so on).

Appendix B

Host Computer Agent Communication Protocol and Client Programs

This appendix describes the communication protocol used by the host computer interface, the standard client program that can be used to interact with the FPGA, and a customized client program that has been used to demo the GraphGen convolutional neural network. The CoRAM++ programming environment currently supports a serial port running at 115,200 BPS on the Terasic DE4, and up to 500,000 BPS on the Xilinx ML605 board (which uses a USB based rather than physical serial port). It also supports communications at full Ethernet line speed on the Digilent Zedboard and Xilinx ZC706 boards with Zynq FPGAs. On Zynq-based FPGAs, the hard ARM cores and ethernet controller on this chip software running on the ARM cores to perform data transfers.

Token	Definition	Comment
Host communication =	**message***	Host communication occurs through an unbounded sequence of messages, all initiated by the host computer.
message =	**nullmessage** |**readdata** |**writedata** |**compute** |**writeparam** |**readparam**	There are 6 types of messages.
nullmessage =	**ready ready**	A null communication is a handshake that allows the host computer to ensure that the communication protocol is working.
readdata =	**read read** address size **reading data ready**	Read *size* bytes of data from *address* on the FPGA to the host system. *Address* and *size* are 4 byte little endian integers.
writedata =	**write write sizeoffset writing data ready**	Write *size* bytes of data data from the host system to address *address* on the FPGA. *Address* and *size* are 4 byte little endian integers.
compute =	**compute compute send8bytes** computecycles **ready**	Instruct the FPGA to compute and return compute cycles, which is sent as an 8 byte integer value in little endian order.
writeparam =	**hosttosys hosttosys** paramindex param **ready**	Write an little endian 8 byte parameter *param* to the application, indexed by the single byte *paramindex*.
readparam =	**systohost systohost** paramindex **send8bytes** param **ready**	Read an little endian byte parameter *param* or debug value from the application, indexed by the single byte *paramindex*.
ready=	0x00	A single byte token.
read =	0x01	A single byte token.
reading =	0x02	A single byte token.
write =	0x03	A single byte token.
writing =	0x04	A single byte token.
compute =	0x05	A single byte token.
send8bytes =	0x06	A single byte token.
hosttosys =	0x07	A single byte token.
systohost =	0x08	A single byte token.

Table B.1: Messages used by the host computer interface (in BNF). Data sent from the host system to the FPGA is shown in blue, and data sent from the FPGA to the host system is shown in teal. **Bold** tokens are defined in the table below the line in which they are presented in bold.

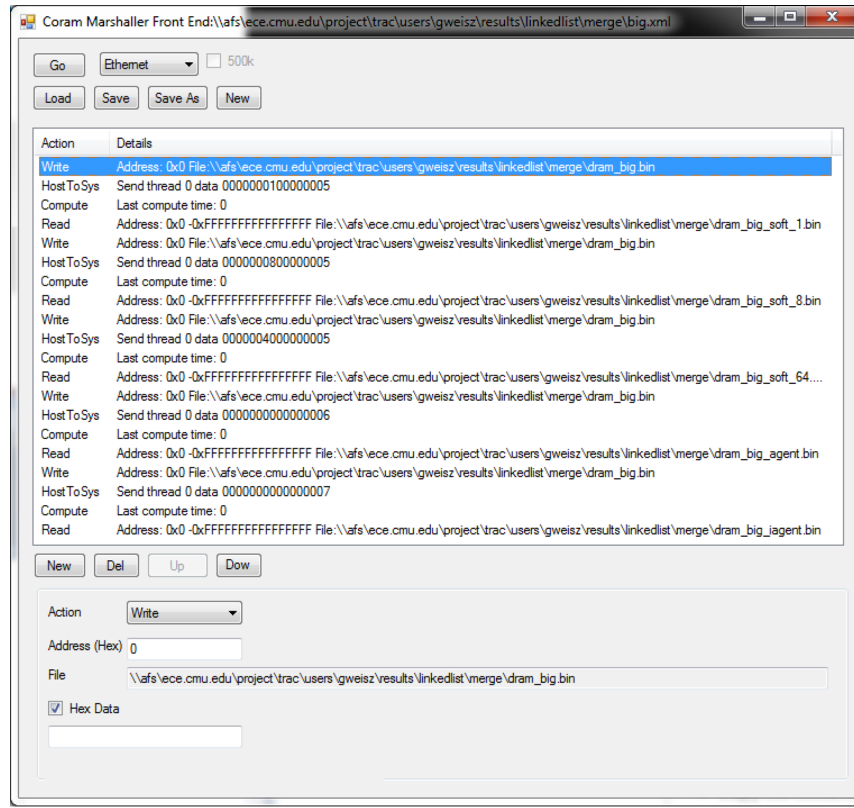


Figure B.1: Screen capture of the host computer interface companion program for Windows.

B.1 Host Computer Interface Communication Protocol

The host computer interface uses the same byte-level communication protocol regardless of the FPGA board and physical transportation medium in use. Table B.1 defines this communication protocol.

B.2 Standard Host Computer Interface Client Program

The client side of the host computer interface is GUI based C# program. Figure B.1 presents a screenshot of this program. This program was first created while working on the

C-to-CoRAM project [81]. This particular screen capture was taken while running linked list traversal experiments as discussed in Section 7.3.

This program is intended to be universal, for any application that does not have much interaction with the host system. The application can write data to FPGA DRAM from a file (or statically defined hex string), read data from FPGA DRAM to a file, write run-time configuration data and read information from the application, and trigger execution and collect run-time in cycles.

While designed for Microsoft Windows, the host program uses no Windows specific features and should be able to run on the Mono .net implementation, and uses an extremely simple line protocol over a serial port or TCP socket, so could easily be replaced by a different application.

B.3 Custom Host Computer Agent Client Program for the GraphGen Convolutional Neural Network Demo

Since the communication protocol used by the host computer interface is well-defined, it is possible to support custom client programs for particular applications. One example of a custom program that interfaces with the host computer interface is the GraphGen Convolutional Network Demo, shown in Figure B.2. This program was used for a live demo at FCCM 2014, and needed more interactive communication with the FPGA. The demo shows the GraphGen-based convolutional neural network application performing handwriting recognition described in Section 7.4. When the program starts up, it loads the FPGA DRAM with default data, but uploads new images to the FPGA, triggers handwriting recognition, and reads the compute time and recognition result in response to user

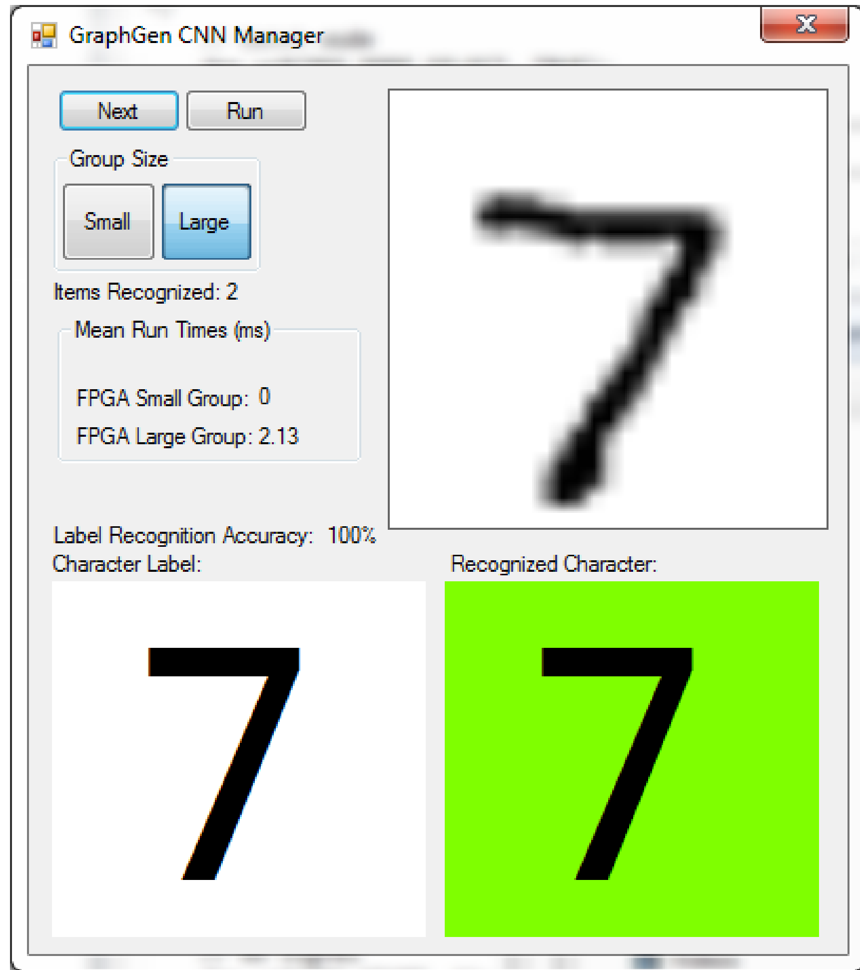


Figure B.2: Screen Capture of the custom host program for the GraphGen Convolutional Neural Network

input. In this particular capture, the application has sent a fuzzy image of a "7" from the MNIST [55] handwriting recognition data set, which is labeled, and the ZedBoard-based application has recognized the number in 2.13 ms. The "Recognized Character" pane is shown with a teal background, indicating that the application did recognize the correct character.

References

[1] Publicly available CoRAM implementation. http://www.ece.cmu.edu/ coram. 71

[2] CPU Clock Frequency by year at the Stanford CPU database. http://cpudb.stanford.edu/visualize/clock_frequency. 1

[3] Compressed Sparse Row Format. http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html. 23

[4] LZRW1 Compression Core. http://opencores.org/project,lzrw1-compressor-core. 128

[5] Top 500 supercomputing systems in November2014. http://www.top500.org/lists/2014/11/. 1

[6] Mohamed S. Abdelfattah and Vaughn Betz. The Case for Embedded Networks on Chip on Field-Programmable Gate Arrays. *IEEE Micro*, 34(1):80–89, 2014. ISSN 0272-1732. 19, 97

[7] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on*

Field Programmable Gate Arrays, FPGA '11, pages 25–28, 2011. 20

[8] B. Akin, F. Franchetti, and J.C. Hoe. Understanding the Design Space of DRAM-Optimized Hardware FFT Accelerators. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 248–255, June 2014. 98, 99, 100

[9] Altera. Altera Annual Report (2012 Form 10-K). http://www.sec.gov/Archives/edgar/data/768251/000076825113000008/altera10k12312012.htm, . 13

[10] Altera. Altera support solution for memory corruption with ALTMEMPHY. http://www.altera.com/support/kdb/solutions/rd03092007_117.html, . rd03092007_117. 140

[11] Altera. Resolution to critical warnings when using the second DRAM controller on the Terasic DE4. http://www.alteraforum.com/forum/archive/index.php/t-30754.html. 140

[12] Altera. Altera C2H Compiler. http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf, . 17

[13] Altera. Altera NIOS Soft Processor. http://www.altera.com/devices/processor/nios2/ni2-index.html, . 14

[14] Altera, Inc. Arria 10 Device Overview, September 2014. AIB-1023. 14, 15, 141

[15] Altera, Inc. Stratix IV Device Handbook, Volume 1, September 2014. SIV5V1-4.6. 66

[16] Altera, Inc. Quartus II Handbook, May 2015. 18

[17] ARM. AXI Bus Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html. 18

[18] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 201–210, Mar 1997. 129

[19] Peter Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction Set Metamorphosis: Compiler and Architecture. *IEEE Computer*, 26:11–18, 1993. 13

[20] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012. 17, 135

[21] Jason D. Bakos. High-Performance Heterogeneous Computing with the Convey HC-1. *Computing in Science Engineering*, 12(6):80 –87, November-Dececember 2010. 13

[22] Tony M. Brewer. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro*, 30(2):70–79, 2010. ISSN 0272-1732. 120

[23] Bruce Wile. Coherent Accelerator Processor Interface(CAPI) for POWER8 Systems, September 2014. 1

[24] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the*

19th ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '11, pages 33–36, 2011. 18

[25] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 245–254, 2013. 13

[26] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP '08, pages 101–107. IEEE Computer Society, 2008. 1, 13

[27] Jungwook Choi and R.A. Rutenbar. Hardware implementation of MRF map inference on an FPGA platform. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 209–216, 2012. 124

[28] Eric S. Chung and James C. Hoe. High-Level Design and Validation of the BlueSPARC Multithreaded Processor. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(10):1459–1470, Oct 2010. 17

[29] Eric S. Chung and Michael K. Papamichael. ShrinkWrap: Compiler-Enabled Optimization and Customization of Soft Memory Interconnects. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 113–116, 2013. 24, 65, 97

[30] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing : Does the Future Include Custom Logic , FPGAs , and GPG-

PUs? *International Symposium on Microarchitecture (MICRO-43), Atlanta, GA, 2010*, pages 225–236, 2010. 1, 13

[31] Eric S. Chung, James C. Hoe, and Ken 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, 2011. 7, 17

[32] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and Evaluation of the CoRAM Memory Architecture for FPGA-Based Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, 2012. 8, 22, 65, 71

[33] J. Coole, J. Wernsing, and G. Stitt. A traversal cache framework for fpga acceleration of pointer data structures: A case study on barnes-hut n-body simulation. In *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, pages 143–148, Dec 2009. 21

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848. 45, 53

[35] Intel Corporation. *Altera Changes the Game for Floating Point DSP in FPGAs*. June 2014. URL https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy. 16, 38

[36] Kattamuri Ekanadham, Jessica Tseng, and Pratap Pattnaik. IBM PowerPC Design in

Bluespec. In *Technical Report RC24706*, 2008. 17

[37] eLinux.org. Android on the ZedBoard. http://elinux.org/Zedboard_Android. 140

[38] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The LEAP FPGA Operating System. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014. doi: 10.1109/FPL.2014.6927488. 17, 20

[39] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 175–184, 2012. 20

[40] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56, 2012. 1, 13

[41] Prabhat K. Gupta. Xeon+FPGA Platform for the Data Center. Presented at the Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic, June 2015. 119

[42] Mark Hachman. Moore's law slows and so does intel, adding a third 14-nm 'kaby lake' chip to roadmap, 2015. URL http://www.pcworld.com/article/2948587/components/moores-law-slows-and-so-does-intel-adding-a-third-14-nm-kaby-lake-chip-to-roadmap.html. 1

[43] Malay Haldar, Anshuman Nayak, Alok Choudhary, and Prith Banerjee. A system for synthesizing optimized FPGA hardware from MATLAB. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 314–319. IEEE Press, 2001. 18

[44] Scott Hauck, Katherine Compton, Ken Eguro, Mark Holland, Shawn Phillips, and Akshay Sharma. Totem: Domain-Specific Reconfigurable Logic. *IEEE Transactions on VLSI Systems*, pages 1–25, 2006. 18

[45] John R. Hauser and John Wawrzynek. Garp: a MIPS Processor with a Reconfigurable Coprocessor. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12–21, 1997. 13

[46] Clint Hilton and Brent Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *Computers and Digital Techniques, IEE Proceedings -*, 153(3):181–188, 2006. 19

[47] T. Hussain, O. Palomar, O. Unsal, A. Cristal, and M. Ayguade, E. anfod Valero. Advanced Pattern Based Memory Controller for FPGA based HPC applications. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, July 2014. 21

[48] Impulse Accelerated. Impulse CoDeveloper. `http://www.impulseaccelerated.com/products.htm`. 17

[49] Intel. Intel Stellarton. `http://ark.intel.com/products/codename/42360/Stellarton`. 16

[50] Robert Kirchgessner, Alan D. George, and Herman Lam. Reconfigurable Comput-

ing Middleware for Application Portability and Productivity. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 211–218, 2013. 20

[51] Vladimir Kolmogorov. Convergent tree-reweighted message passing for energy minimization. In *Proc. Intl Workshop Artificial Intelligence and Statistics*, 2005. 124

[52] William V. Kritikos, Andrew G. Schmidt, Ron Sass, Erik K. Anderson, and Matthew French. Redsharc: A programming model and on-chip network for multi-core systems on a programmable chip. *Int. J. Reconfig. Comp.*, 2012, 2012. 19

[53] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 64, 65

[54] Maysam Lavasani, Larry Dennison, and Derek Chiou. Compiling High Throughput Network Processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 87–96, 2012. 18

[55] Y. LeCun, C. Cortes, and C. J. C. Burges. *The MNIST database of handwritten digits*. 149

[56] Yann LeCun, Lèon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 127

[57] Maxeler Technologies. MaxCompiler White Paper, February 2011. 20

[58] Mentor Graphics . Catapult-C. `http://www.mentor.com/esl/catapult/`

overview. 17

[59] Mentor Graphics. Handel-C. http://www.mentor.com/products/fpga/handel-c. 17

[60] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012. 18, 71

[61] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012. 41, 95, 98

[62] Yehudit Mond and Yoav Raz. Concurrency control in b+-trees databases using preparatory operations. In *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden.*, pages 331–334, 1985. 54

[63] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38 (8):114–117, April 1965. ISSN 0018-9219. 1

[64] G. M. Morton. A Computer Oriented Geodetic Data Vase and a New Technique in File Sequencing. Technical Report Ottawa, Ontario, Canada, 1966. 50

[65] MyHDL. www.myhdl.org. 17

[66] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004. 17, 65, 66, 71

[67] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28, May 2014. 121, 127

[68] NVIDIA. The NVIDIA CUDA language. http://www.nvidia.com/object/cuda_home_new.html. 5

[69] M. ONeill. *Neural Network for Recognition of Handwritten Digits*. CodeProject Website. 127

[70] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42, 2009. 18

[71] Michael K. Papamichael and James C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing NOCS in the Context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, 2012. 19

[72] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft connections: Addressing the Hardware-Design Modularity Problem. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 276–281, July 2009. 43

[73] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990. 115

[74] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014. 1, 21

[75] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47:7–42, 2002. 125

[76] Siddhartha Sen and Robert Endre Tarjan. Deletion without rebalancing in multiway search trees. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 832–841, 2009. 54

[77] James E. Smith. Decoupled Access/Execute Computer Architectures. In *ACM Transactions on Computer Systems*, pages 112–119, 1984. 7

[78] David Barrie Thomas, Lee Howes, and Wayne Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 63–72, 2009. 1, 13

[79] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing*

Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pages 127–134, May 2010. 17, 25

[80] Matthew A. Watkins and David H. Albonesi. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 497–508. IEEE Computer Society, 2010. 13

[81] Gabriel Weisz and James C. Hoe. C-to-CoRAM: Compiling Perfect Loop Nests to the Portable CoRAM Abstraction. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 221–230, 2013. ISBN 978-1-4503-1887-7. 18, 24, 57, 69, 148

[82] Gabriel Weisz, Eriko Nurvitadhi, and James C. Hoe. Graphgen for coram: Graph computation on fpgas. In *The Third Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013)*, December 2013. 121

[83] R.N. Williams. An extremely fast ziv-lempel data compression algorithm. In *Data Compression Conference, 1991. DCC '91.*, pages 362–371, Apr 1991. 128

[84] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George A. Constantinides. MATCHUP: memory abstractions for heap manipulating programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 136–145, 2015. 119

[85] Xilinx. Vivado High Level Synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/`, . 17

[86] Xilinx. Xilinx Zynq All-Programmable SoC. http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm, . 14, 15, 38

[87] Xilinx. Design Advisory Zynq-7000 PS DDR Controller - DDR IO's are not properly configured in ISE/EDK and Vivado 2013.3 and earlier. http://www.xilinx.com/support/answers/60454.html, . AR #60454. 141

[88] Xilinx. MicroBlaze Processor Reference Guide. 2009. 14

[89] Xilinx, Inc. Virtex-6 Family Overview, January 2012. v2.4. 66

[90] Xilinx, Inc. Embedded System Tools Reference Manual, June 2013. 18

[91] Xilinx, Inc. Zynq-7000 All Programmable SoC Overview, October 2014. v1.7. 66, 140

[92] Xilinx, Inc. AXI DMA v7.1 LogiCORE IP Product Guide, April 2015. 19

[93] Hsin-Jung Yang, K. Fleming, M. Adler, and J. Emer. Optimizing under abstraction: Using prefetching to improve fpga performance. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013. 20

[94] Hsin Jung Yang, Kermin Fleming, Michael Adler, and Joel Emer. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 117–124, May 2014. 20

[95] Yi-Hua Edward Yang and Viktor K. Prasanna. High throughput and large capacity pipelined dynamic search tree on fpga. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA

'10, pages 83–92, New York, NY, USA, 2010. ACM. 85

[96] ZedBoard.org. ZedBoard Pricing. http://zedboard.org/buy. 140

[97] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977. 128

[98] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, 2013. 18