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

Gabriel Weisz
Carnegie Mellon University
Pittsburgh, PA, USA
gweisz@cs.cmu.edu

James C. Hoe
Carnegie Mellon University
Pittsburgh, PA, USA
jhoe@ece.cmu.edu

***Abstract*—Facilitating DRAM access is an essential part of an application programming environment for FPGA computing. Existing FPGA application programming environments primarily focus on support for simple, regular memory access patterns, such as block copy and streaming. This paper presents CoRAM++, a programming environment for FPGA computing that is based on an extensible set of data-structure-specific memory interfaces. CoRAM++ supports complex data structures, such as multi-dimensional arrays and linked lists, in addition to simple data access patterns. CoRAM++ defines an appropriate application-level interface for each supported data structure, and provides a specialized soft-logic implementation of the supporting datapath to memory We evaluated the effectiveness of the CoRAM++ data-structure-specific approach in three application scenarios based on streams, multi-dimensional arrays and linked lists. Our results show that the CoRAM++ programming environment can offer convenient application-level interfaces without penalizing DRAM access performance.**

I. Introduction

Motivation. All but the most trivial FPGA computing applications require accessing data in external DRAM. In the past, FPGA computing applications interacted directly with low-level DRAM interfaces, which were tedious to use. Direct interaction with the low-level DRAM interfaces also locked the application logic to the non-portable wire-level and timing-level details of the targeted FPGA device and board environment. More recently, FPGA application programming environments have layered simplifying application-level interfaces, supported by soft-logic shims, over native DRAM and I/O interfaces [1]–[4]. Their abstracted application-level interfaces simplify interaction with DRAM and enable application portability. However, existing programming environments have predominantly focused on support for simple, regular memory access patterns, such as block copy and streaming. This paper presents the CoRAM++ programming environment, which efficiently and extensibly supports complex data structures, such as multi-dimensional arrays and linked lists, in addition to simple data access patterns.

Relationship to CoRAM. At the highest level, CoRAM++ follows the previously proposed CoRAM [2] approach of decomposing an FPGA computing application into separate computation and control components (see Figure 1). The goal of the CoRAM approach is to shield hardware computation kernels from the complex and platform-specific interactions with external DRAM. The hardware computation kernels are isolated to only interact with on-chip SRAM blocks (within the application-level interface) for data input and output. A multi-threaded C-like language—ultimately compiled to finite state machines (FSMs)—supports the application developer in expressing control sequencing: (1) the data movements between the off-chip DRAM and the on-chip SRAM; and (2) the invocations of the hardware kernels. An application can comprise multiple control threads and hardware kernels.

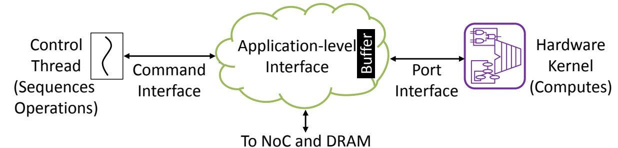

Fig. 1: CoRAM/CoRAM++ application decomposition into separate compute and control, providing locally buffered data to a hardware computation kernel.

To support this enforced decomposition, CoRAM divides application-level interfaces into two parts: (1) the wire-level port interfaces used by hardware kernels for accessing locally buffered data; and (2) the set of commands used by control threads for orchestrating data movements between DRAM and the hardware kernels. The original CoRAM programming environment supports a block copy application-level interface. The hardware kernels use wire-level port interfaces that specifically connect to local SRAM blocks, and the control threads use commands that transfer small, contiguous blocks of data between DRAM and the SRAM blocks. As originally proposed, CoRAM's application-level interface was meant to serve as a universal primitive that is hardened in a future FPGA in order to efficiently support all memory access patterns.

CoRAM++ Data-Structure-Specific Interfaces. CoRAM's application-level interface can be realized in today's FPGAs through a soft-logic implementation of the interface, but with penalties in terms of both logic resource overhead and performance [5]. CoRAM++ reduces these penalties by making better use of the reconfigurability of the FPGA fabric to provide a more customized datapath to memory. This improves performance, reduces resource overheads, and offers increased convenience to the application developer. CoRAM++ achieves these goals through an extensible library of data-structure-specific data transport modules, which each defines an appropriate application-level interface.

For example, streaming a block of data from memory is a common operation. A CoRAM++ application-level interface

for streaming provides control threads with specialized 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 hardware kernels can only read (or write) 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 streamlined soft-logic implementations 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.

Paper Outline. Following this introduction, Section II highlights contemporary FPGA computing programming environments and their support for accessing DRAM. Section III presents the workings of the CoRAM++ data-structure-specific programming environment in greater detail. Section IV delves into the construction of the CoRAM++ library of application-level interfaces. Section V presents our evaluation of the CoRAM++ data-structure-specific approach in three application scenarios based on streams, multi-dimensional arrays, and linked lists. Finally, Section VI concludes.

II. Background and Related work

Section I introduced the original CoRAM programming environment and its application-level interface for DRAM [2]. Below we survey other FPGA computing programming environments, which have primarily focused on supporting simple, regular memory access patterns such as block copy and streaming.

Altera and Xilinx both provide "system builder" tools to assist in system integration ([6], [7]), which use buses to connect DRAM interfaces, application components, and processor cores. These tools support load-store and streaming interfaces. and allow application developers to manually incorporate descriptor-based DMA engines [8].

Maxeler MaxCompiler [9] 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.

Redsharc [3] supports hardware kernels with application-level interfaces for streaming and block copy. Like CoRAM and CoRAM++, Redsharc applications are decomposed into separate control and compute components. In Redsharc, global control is managed by software running on an embedded processor.

RCMW [4] presents a virtualized FPGA board environment which is portable across different physical FPGA boards through the use of a shim layer. RCMW's virtual environment has support for streaming and block copy.

LEAP [1] provides an operating environment in which the hardware kernels interface to "scratchpad" memory through latency-insensitive channels (using a load-store interface). The capacity of a scratchpad can exceed what can be held on-chip in SRAM and is backed up by off-chip DRAM. An automatic cache can be instantiated to serve fast local copies; performance can be further enhanced by instantiating an automatic cache prefetcher. Multiple hardware kernels—including those on different FPGA boards—can share a coherent view of the same virtualized scratchpad. LEAP provides a complete, generic abstraction over the base DRAM subsystem, but does not take advantage of optimizations that may be possible when the application's data access patterns are known.

Most similar to CoRAM++ in motivation is APMC [10], 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 [11] to store the results of a data structure traversal for later reuse.

III. CoRAM++ Data-Structure-Specific Application-Level Interfaces

A. Argument for Specializing in a Soft-Logic Context

The application-level interface for memory accesses provided by the original CoRAM programming environment is based on a block copy paradigm. Hardware kernels interface with on-chip SRAM blocks. CoRAM control threads issue commands to cause transfers of small, contiguous blocks of data between off-chip DRAM and on-chip SRAM.

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

TABLE I: Currently supported data-structure-specific application-level interfaces.

Interface	Description	Port interfaces for hardware kernels	Commands for control threads
Block Copy	Copies blocks of data between DRAM and local SRAM blocks.	enable, address, data_out[], write, data_in[]	write_ram(global_addr,local_addr,bytes) read_ram(global_addr,local_addr,bytes) check_status(transaction_tag)
Read Stream	Streams data from DRAM to a hardware kernel.	empty, data_out[], deq	read_stream(addr,size) check_status()
Write Stream	Streams data from a hardware kernel to DRAM.	full, data_in[], enq	write_stream(addr,size) check_status()
Mult-dimensional Array Reader	Transfers array data from DRAM to a hardware kernel.	empty, data_out[], deq	read_array(addr,dimension,layout,permuter) read_items(addr,dimension,layout,permuter,first,count) check_status()
Mult-dimensional Array Writer	Transfers array data from DRAM to a hardware kernel.	full, data_in[], enq	write_array(addr,dimension,layout,permuter) write_items(addr,dimension,layout,permuter,first,count) check_status()
Linked List	Traverses and merges linked lists, streaming linked list data to a hardware kernel when traversing.	empty, data_out[], deq	traverse_list(addr) merge_lists(addr1,addr2) check_status()

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 falls short 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 III-B, 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).

B. Supported Data Structures and their Interfaces

This paper presents a subset of select data structure examples to demonstrate the concept of CoRAM++. We intend the CoRAM++ data structure library to be extensible—over time it will grow to include other key data structures that are fundamental to computing [12] or critical to highly-valued application domains. The currently supported data structures and their application-level interfaces are summarized as follows:

- **Block Copy:** Copies small contiguous blocks of data between DRAM and local storage. Allows random access by the hardware kernel to data within a block. This is the same interface supported by the original CoRAM application-level interface.
- **Read/Write Stream:** Transfers a continuous stream of data to or from consecutive DRAM locations. Allows sequential access by the hardware kernel to a single data element at a time.
- **Multi-dimensional Array:** Streams a sequence of array elements along any array dimension. Requires that the size and dimensions of the array are defined at compile time. Supports tiled layout optimizations. Allows sequential access by the hardware kernel to a single data element at a time. Includes support for permutation engines that reorder data streams, which can be selected at run time.
- **Linked list:** Traverses and merges linked lists. Contains a hardware linked list engine;[1] Requires compile-time definition of the linked list data structure. Allows sequential access by the hardware kernel to a single data element at a time when traversing the linked list.

As mentioned in Section I, each application-level interface in CoRAM++ has two parts: (1) the wire-level port interfaces that are attached to the hardware kernels for accessing locally buffered data; and (2) the set of commands that are available to the control threads to control data movements between DRAM and the hardware kernels. Table I provides a summary of the two-part application-level interfaces for the currently supported data structures. Except for block copy, the application-level interfaces listed in in Table I all provide FIFO-style wire-level port interfaces to hardware kernels. They could easily be modified to provide SRAM-style hardware port interfaces that allow random access to elements in a block of data.

In addition to the memory interfaces described above, a special control-thread-side **Scratchpad** application-level inter-

[1] The hardware linked list engine follows pointers and collects payload data for consumption by the application or attached merge engine. It contains a small direct mapped cache and can be directly connected to a DRAM interface.

face allows control threads to access data from DRAM, and can be configured to fetch blocks of data in order to take advantage of data locality. Another control-thread-side **Host computer** application-level interface allows control thread to manage data transfers between a host computer and the FPGA's attached DRAM. The host computer interface can also transfer run-time parameters, debugging information, and benchmarking statistics. Finally, a **Channel** interface transmits short messages between a hardware kernel and a control thread, between two hardware kernels, or between two control threads.

C. Code Example

This section illustrates a CoRAM++ streaming computation code example. A CoRAM++ application developer creates an application by: (1) designing the hardware kernels to perform computation; (2) selecting the appropriate application-level interface "agents" for the data structures; and (3) writing the software control threads that sequence computation and communication. We use the term "agents" to refer to the underlying implementation objects that support the application-level interfaces. Listing 1 provides the Verilog HDL excerpt of a hardware kernel for a discrete Fourier transform (DFT) kernel with streaming input and output interfaces. Listing 2 provides its corresponding control thread.

In Listing 1, lines 7-9 declare this kernel's control thread, referring to it by name "dft_thread"; this effectively instantiates a FSM that carries out the control sequencing in Listing 2. Lines 11-14 (or 15-19) instantiate a FIFO-like read (or write) port corresponding to the streaming agent s_in (or s_out). The wire-level signals from the read and write ports are connected to a normal Verilog module for DFT (lines 21-25). Matching valid and stall signals provide handshaking between the DFT module and the streaming input and output streaming agent port interfaces.

Listing 1: Hardware kernel for CoRAM++ Streaming 64-input DFT.

```
1 // dft.v: Hardware kernel code
2 module dft_kernel();
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  // Handle to the control thread
7  ControlThread #(.FNAME("dft_thread"),
8   .PARAMS("IN_ADDR=0;OUT_ADDR=0x800000;
9   SIZE=8192")) appThread;
10 // Streaming Agents for data transfers
11 ReadStream#(.ID(1),.TID("dft_thread"),
12  .WIDTH(256))
13  inData(.notEmpty(in_rdy),
14   .first(dft_in),.deq(in_rdy&&dft_rdy);
15 WriteStream#(.ID(2),.TID("dft_thread"),
16  .WIDTH(256))
17  outData(.data(out_data),
18   .notFull(out_rdy),
19   .enq(dft_done&&out_rdy));
20 // DFT IP core performing computation
21 DFT#(.SIZE(64),.PRECISION(64)) kernel(
22  .in(dft_in), .out(dft_out),
23  .out_valid(dft_done),
24  .issue(in_rdy&&dft_rdy),
25  .stall(dft_done&&!out_rdy);
26 . . . .
27 endmodule
```

Listing 2: Control Thread for CoRAM++ Streaming 64-input DFT.

```
1 // dft.c: Control thread code
2 #include "StreamAgent.h"
3 void dft_thread(int IN_ADDR,
4   int OUT_ADDR, int SIZE) {
5   Agent s_in=getStreamAgent(1);
6   Agent s_out=getStreamAgent(2);
7   . . . .
8   s_in.read_stream(IN_ADDR,SIZE);
9   s_out.write_stream(OUT_ADDR,SIZE);
10  while (STREAM_DONE !=
11   s_out.check_stream()) {}
12  . . . .
13 }
```

Listing 2 shows a very simple example of a C-syntax control thread. Lines 5 and 6 connect to the two streaming "agents" with unique IDs 1 and 2 as arguments; one agent provides input data, and the other transfers output data to DRAM. The "read_stream" (or "write_stream") command is invoked in line 8 (or 9) to establish a read (or write) stream of length SIZE starting from address location IN_ADDR (or OUT_ADDR). The commands are non-blocking, and SIZE, IN_ADDR, and OUT_ADDR could be supplied at run time through the host computer interface. Lines 10-11 have a short loop that repeatedly calls a non-blocking check_stream function until the "s_out" data transfer has completed.

In general, control threads can use the common control flow structures in the C language, including dynamically-bounded loops and function calls, and can access data from DRAM using the scratchpad application-level interface. When mapping a control thread to a FSM, CoRAM++ compiles function calls through inline expansion when possible, but can also support function calls using a fixed-depth hardware stack when inline expansion is not possible. As is the case with the original CoRAM, a CoRAM++ application can employ multiple control threads and hardware kernels. One control thread can manage multiple ports, but a port can only be managed by one control thread.

IV. Implementing the CoRAM++ Interfaces

After an application has been specified as its hardware kernels, control threads, and the desired application-level interfaces (such as the example in Section III-C), the CoRAM++ compiler combines the application components into an implementation ready for final processing by FPGA vendor tools. The CoRAM++ compiler currently targets commodity FPGAs from both Altera and Xilinx, and allows applications to specify

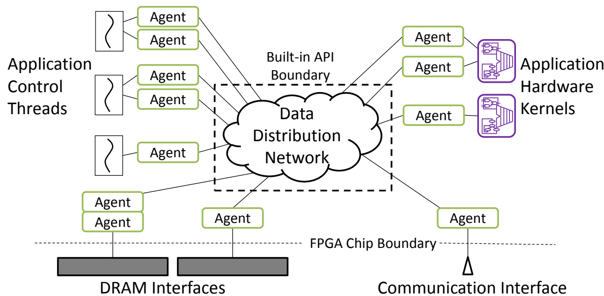

Fig. 2: System-level design of a CoRAM++ FPGA computing application.

how DRAM interfaces and other communication interfaces are mapped into a global address space.

We call the hardware modules implementing the application-level interfaces "agents". Control threads connect to agents through a "command" interface, and hardware kernels connect to agents through a "port" interface. In general, port interfaces can include any combination of wire-level ports, including multi-wire compound interfaces such as FIFOs, SRAMs, and CAMs. When compiling control threads to FSMs, the CoRAM++ compiler converts agent-supported function calls into messages sent over these hardware command interfaces, which might be directly connected to the hardware port of an agent, or might be transferred to an agent's control thread[2] that performs a sequence of operations in response to the message.

CoRAM++ agent control threads are key to allowing CoRAM++ to support the most convenient application-level interfaces possible. Applications can send commands with long-running effects to agents in a "fire-and-forget" fashion. This "fire-and-forget" execution model allows agents to independently perform complex data transfers on behalf of application components. Although callback functions and manually created control threads can also be used to manage complex data transfers, a "fire-and-forget" mechanism provides the most convenience to the application developer.

Figure 2 depicts the system-level design of an FPGA computing application utilizing CoRAM++. The figure depicts a number of different hardware kernels and control thread FSMs. Various agents are inserted between the application components (kernels and control FSMs) and the FPGA-specific base infrastructure for accessing DRAM and for on-chip data transport.[3] This figure does not depict the low-level implementation details of agents, such as agent control threads or the point-to-point connections created by the Channel agent.

Agents created for CoRAM++ do not see the FPGA's device-specific base infrastructure directly, but are instead supported by a thin hardware virtualization layer, which provides interfaces for:

[2]Agents themselves are developed within the CoRAM++ programming environment and therefore can utilize the control thread abstraction to specify control FSMs.

[3]While hardened DRAM interfaces are increasingly common, the CoRAM++ programming environment is anticipating that fast and efficient hardwired Network-on-Chip (NoC) data transport will also become available in future FPGAs [13]. Until then, CoRAM++ adopts a soft-logic NoC [14] as the means for systematic data transport between distributed components.

TABLE II: FPGA boards used in our evaluation.

FPGA Board	Terasic DE4	Xilinx ML605	Xilinx ZC706
FPGA	Altera EP4SGX530	Xilinx LX240T	Xilinx XC7Z045
Hard CPU Cores	None	None	2×ARM A9
Logic Cells	531,200	241,152	350,000
Block Memory	3.4 MB	1.8 MB	2 MB
Hard DSP blocks	1,024	768	900
DRAM Bandwidth	2×6.4 GB/s	6.4 GB/s	12.8 GB/s (Fabric only) 4.3 GB/s (Shared)
FPGA Vendor Tool	Quartus 14.0	ISE 14.7	ISE 14.7

- **Data transfers** over the NoC between DRAM (or a communication interface mapped into an address space) and agent components.
- **Messages** between agents and directly connected hardware kernels and control threads.
- **Direct connections** between agents and interfaces for external DRAM and I/O.

This hardware virtualization layer makes it easier to port the CoRAM++ interface library to different FPGAs. This hardware virtualization layer also makes it possible to port the CoRAM++ interface library to take advantage of the programming environments discussed in Section II, for example to make use of LEAP's automatic cache and prefetching mechanisms.

Most of the time, agents interact with the control thread FSMs or the hardware compute kernels in order to support application-level interface functionalities. Some memory access patterns call for access accelerator agents that speed up interaction with DRAM and I/O interfaces. Multiple accelerator agents can be attached to a DRAM or I/O interface in a chain. These accelerator agents are able to manipulate those interfaces directly to minimize access latency. Accelerator agents can also incorporate private caches or scratchpads. Accelerator agents are essential for delivering good performance on pointer-based irregular memory access patterns such as linked lists, trees, and graphs.

V. Evaluation

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. We evaluated two application scenarios that were bandwidth bound, and one that was latency bound. This evaluation did not investigate compute bound application scenarios, but discusses the resource overheads that would affect compute-bound applications.

Table II describes the characteristics of the FPGA development boards used in our evaluation. The Xilinx boards were only used in the latency bound application scenario, and experiments using the ZC706 operated on data in the higher-bandwidth fabric-only DRAM interface.

A. Streaming Accesses

Our first application scenario shows that the run time and resource overheads of CoRAM++ are acceptable for simple streaming accesses. These experiments computed a 1D DFT

using a hardware kernel very similar to that of Listing 1 and control thread code very similar to that of Listing 2. We used the host computer agent to manage the computation and benchmark execution.

Our kernel used a Spiral-generated [15] 64-input double precision DFT engine running at 200 MHz, streaming input data from one DRAM interface and streaming output data to the other DRAM interface on the DE4. Each experiment computed on multiple contiguous data sets as a long stream.

Figure 3 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 run-time 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 III breaks this resource utilization down by component, showing that the DRAM controllers and NoC consume significant resources. An FPGA (such as [13]) that hardens these components would incur less overhead. ShrinkWrap [16] could also reduce NoC-related resource requirements for application scenarios that are less bandwidth constrained.

Fig. 3: Streaming double precision 64-input DFT performance and DRAM controller microbenchmark results on the DE4.

TABLE III: 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

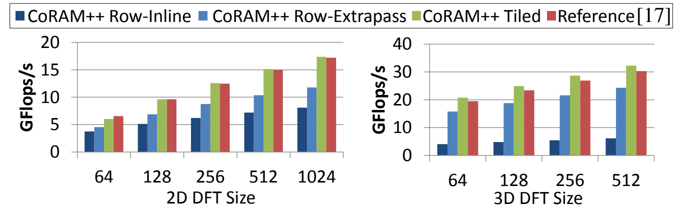

Fig. 4: 2D (left) and 3D (right) DFT performance.

TABLE IV: 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 [17] 1024×1024 2D	27	47	35

B. Multi-dimensional Array Accesses

Our second 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 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 [15].

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.

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 [17]. 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 4 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 [17], which also used a tiled data layout throughout the computation.

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.

Table IV 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.

C. Linked List Accesses

Our final 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.

This application scenario traversed linked lists and merged sorted linked lists in place. Each experiment ran the control threads, kernel, and agents 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:

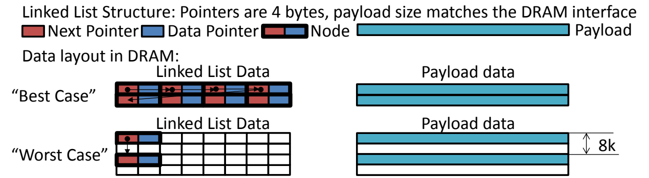

Fig. 5: Linked list configurations

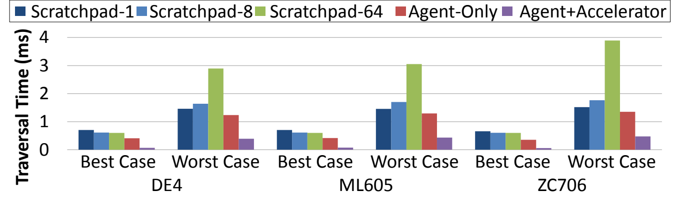

Fig. 6: Linked list traversal performance

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 accesses.
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.

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 5.

Figure 6 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 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

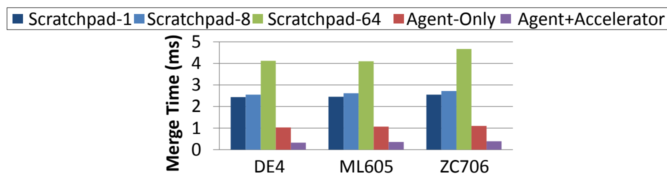

Fig. 7: Sorted linked list merge performance

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.

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.

Figure 7 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 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.

VI. Conclusions and Future Work

As FPGAs grow larger, FPGA applications can afford to expend resources on frameworks that simplify application development and provide application portability. The CoRAM++ framework conveniently and efficiently supports simple and complex data access patterns through a library of data-structure-specific interfaces. Our evaluation demonstrates that CoRAM++ allows applications to saturate the bandwidth provided by DRAM interfaces, match the performance of hand designs, and use data-structure-specific modules that are directly connected to a DRAM interface for up to 5.2× better performance on data-dependent operations.

Future explorations using the CoRAM++ FPGA programming environment will accelerate more data structures including trees and graphs. We will also investigate hardware/software codesign with the hard processor cores and caches of hybrid CPU-FPGA systems, and target other FPGA programming environments such as LEAP rather than specific FPGAs.

Acknowledgments

This work is supported by NSF CCF-1012851. The authors thank Altera, Xilinx, and Bluespec for tools, hardware, and support. The authors thank Berkin Akin for help with the Spiral DFT kernels and permutations.

References

[1] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA Operating System," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, September 2014.

[2] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 2011.

[3] W. V. Kritikos, A. G. Schmidt, R. Sass, E. K. Anderson, and M. French, "Redsharc: A programming model and on-chip network for multi-core systems on a programmable chip." *Int. J. Reconfig. Comp.*, 2012.

[4] R. Kirchgessner, A. D. George, and H. Lam, "Reconfigurable Computing Middleware for Application Portability and Productivity," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, 2013.

[5] E. S. Chung, M. K. Papamichael, G. Weisz, J. C. Hoe, and K. 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*, ser. FPGA '12, 2012, pp. 139–142.

[6] Altera, Inc., "Quartus II Handbook," May 2015.

[7] Xilinx, Inc., "Embedded System Tools Reference Manual," June 2013.

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

[9] Maxeler Technologies, "MaxCompiler White Paper," February 2011.

[10] 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.

[11] 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*, Dec 2009, pp. 143–148.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[13] M. S. Abdelfattah and V. Betz, "The Case for Embedded Networks on Chip on Field-Programmable Gate Arrays," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, 2014.

[14] M. K. Papamichael and J. 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*, ser. FPGA '12, 2012, pp. 37–46.

[15] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 2, 2012.

[16] E. S. Chung and M. 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*, May 2013.

[17] B. Akin, F. Franchetti, and J. 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*, June 2014, pp. 248–255.