CoRAM: An In-Fabric Memory Abstraction for FPGA-Based Computing

Eric S. Chung, James C. Hoe, and Ken Mai
Carnegie Mellon University, Pittsburgh, PA 15213
{echung, jhoe, kenmai}@ece.cmu.edu

ABSTRACT

FPGAs have been used in many applications to achieve orders-of-magnitude improvement in absolute performance and energy efficiency relative to conventional microprocessors. Despite their newfound potency in both processing performance and energy efficiency, FPGAs have not gained widespread acceptance as mainstream computing devices. A fundamental obstacle to FPGA-based computing can be traced to the FPGA's lack of a common, scalable memory abstraction. When developing for FPGAs today, application writers are often directly responsible for crafting the application-specific infrastructure logic that transports data to and from the processing kernels. This infrastructure not only increases design time and effort but will often inflexibly lock a design to a particular FPGA product line, hindering scalability and portability. We propose a new FPGA memory abstraction called Connected RAM (CoRAM) to serve as a portable bridge between the distributed computation kernels and the edge memory interfaces. In addition to improving performance and efficiency, the CoRAM architecture provides a virtualized memory environment as seen by the hardware kernels to simplify application development and to improve an application's portability and scalability. This CARL workshop research overview summarizes our published work in FPGA'11 [4].

1. INTRODUCTION

With power becoming a first-class architectural constraint, future computing devices will need to look beyond general-purpose processors. Among the available computing alternatives today, Field Programmable Gate Arrays (FPGA) have been used in many applications to achieve orders-of-magnitude improvement in absolute performance and energy efficiency relative to conventional microprocessors (e.g., [8, 5, 3]). A recent study [5] further showed that FPGA fabrics can become an effective computing substrate for floating-point intensive numerical applications, even in comparison to GPU technologies.

While accumulated VLSI advances have steadily improved the processing capability of reconfigurable logic, FPGAs have not gained widespread acceptance as mainstream computing devices. A commonly cited obstacle is the difficulty in programming FPGAs using low-level hardware development flows. Beyond that, a more fundamental obstacle to FPGA-based computing can be traced to the FPGA's lack of a common, scalable memory abstraction as seen by application writers. When developing for an FPGA today, a designer has to create from bare fabric not only the application kernel itself but also the application-specific infrastructure logic to support and optimize the transfer of data to and from edge memory interfaces. Very often, creating or using the infrastructure logic not only increases design time and effort but will often inflexibly lock a design to a particular product environment, hindering scalability and portability. Further, the support mechanisms which users are directly responsible for will be increasingly difficult to manage in the future as: (1) embedded SRAMs scale and become more distributed across the fabric, and (2) long-distance interconnect delays become more difficult to tolerate as FPGAs begin to reach unprecedented levels of capacity [13].

The Need for a Common Memory Abstraction. The root cause of many programmability challenges that affect FPGAs in computing today can be attributed to the lack of the most basic standard abstractions that one comes to expect in a general purpose computer—e.g., an ISA, virtual memory, word size definitions, memory modes, address/data formats, etc. From a computing perspective, a common shared abstraction is a critical ingredient for programmability and portability—beyond that, an effective abstraction should also be amenable to scalable implementations and novel optimizations of the underlying hardware.

The abstractions that exist for general-purpose processors today, however, do not readily apply to FPGAs due to their vastly different, non-von Neumann computing characteristics. Rather than coarse-grained processing units, modern FPGAs consist of up to millions of interconnected, fine-grained distributed lookup tables (LUTs) and thousands of embedded SRAMs in a single chip [13]. In serving the common needs of FPGA applications and users, the central goal of this work is to create a re-usable, shared computing abstraction that suits the distributed nature of FPGA-like fabrics—with the particular emphasis on a **proper memory abstraction**. Working under the above premises, the guiding principles for the desired memory abstraction are:

- *The abstraction should present to the user a standard, virtualized appearance of the FPGA fabric, which encompasses reconfigurable logic, its memory interfaces, and the multitude of SRAMs—while freeing application writers from details irrelevant to the application itself.*
- *Over the course of the computation, the abstraction should provide a common, easy-to-use mechanism for controlling the transport of data between memory interfaces and the SRAMs used by the application.*
- *The abstraction should be amenable to scalable FPGA microarchitectures without affecting the architectural view presented to existing applications.*

2. CORAM ARCHITECTURE

Overview and assumptions. The CoRAM architecture

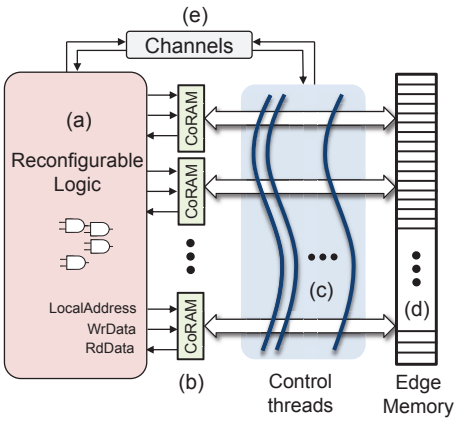

Figure 1: The CoRAM Memory Abstraction.

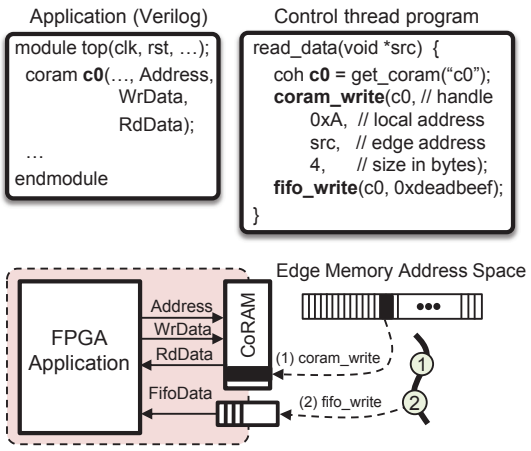

Figure 2: Example Usage of CoRAM.

presents an abstraction that enforces a separation of concerns between computation, data marshalling, and control sequencing within reconfigurable logic. Figure 1 shows the programmer's view of how applications are decomposed when targeting reconfigurable logic with CoRAM support. Beginning from Figure 1(a), applications are first mapped into reconfigurable logic resources available to the user. The CoRAM architecture assumes that reconfigurable logic resources will exist either as stand-alone programmable devices on a multiprocessor memory bus or integrated into a single-chip heterogeneous multicore. Regardless of the configuration, it is assumed that memory interfaces for loading from and storing to a linear address space will exist at the external boundaries of the reconfigurable logic (see Figure 1(d)). These interfaces are implementation-specific and do not affect or alter the CoRAM abstraction—however, they are likely to be supported by conventional memory hierarchy designs, which could include not only dedicated memory controllers but also caches and even full-fledged virtual memory management.

Connected RAM. A key requirement of the CoRAM abstraction is that FPGA applications hosted within reconfigurable logic are not permitted to access physical memory or I/O interfaces directly (or be aware of their details). Applications within reconfigurable logic are instead logically confined by a collection of embedded SRAMs (referred to as CoRAMs) that provide distributed, on-chip storage for application data (see Figure 1(b)). Much like the memory architectures of conventional FPGA fabrics, CoRAMs possess the desirable traits of fabric-embedded SRAM [11]—they have a simple SRAM interface with deterministic access times, are spatially distributed, and can provide high aggregate on-chip bandwidth; they also support composition and flexible aspect ratios. CoRAMs, however, deviate drastically from conventional embedded SRAMs in the sense that the contents of individual CoRAMs are actively managed by a specialized set of programmable logical finite state machines called "control threads" as shown in Figure 1(c).

Control threads. Control threads allow the user to explicitly control data movements between the edge memory address space and the locally-addressed CoRAMs distributed throughout the fabric. Within the CoRAM abstraction, **all** memory accesses from the application to the external environment (e.g., main memory, I/O, etc.) are restricted through the use of CoRAMs, each of which are bound to an associated control thread (a single control thread could, however, manage multiple CoRAMs). The simple example shown in Figure 2 illustrates how an FPGA user would (1) instantiate a CoRAM in their RTL design (left), and (2) program a corresponding control thread to read a single data word from edge memory into the CoRAM (right). The control thread program shown in Figure 2 (right) first acquires a handle to the single instantiated CoRAM (*get_coram("c0")*) and second, executes *coram_write*, which is an operation that performs a 4-byte memory transfer from the edge memory address space to the local CoRAM. To inform the application when the data is ready to be accessed for computation, the control thread sends a token to the FPGA application through an available asynchronous FIFO.

Discussion. As the simple example shows, control threads are high level descriptions of an application's memory access pattern and are programmed using a portable, C-based language abstraction. The use of a high level language for control threads affords an application developer not only simpler but also more natural expressions of control flow and memory pointer manipulations; control threads can also maintain local state to assist in thread activities. Control threads communicate to reconfigurable logic applications through minimal channels as shown in Figure 1(e). If necessary, control threads can also communicate with other threads through a simple message-passing interface (although no global shared memory is allowed). Control threads also cannot perform dynamic memory allocation such as *malloc*.

Each control thread must be reducible to an equivalent finite-state machine (FSM) but in practice may be compiled either to FSMs implemented in reconfigurable logic itself or to software executables hosted on hardwired microcontrollers if available as part of dedicated mechanisms for CoRAMs. Overall, the inefficiencies of a high-level programming language would not directly impede the overall computation throughput because the control threads do not "compute" in the usual sense but are used only to generate requests to memory. Details about possible implementation methods will be covered later in Section 3.

```
/*** CoRAM handle definition ***/
struct {int n_corams; int width;
        int depth; void *addr; ...} coh;

/*** Handle acquisition ***/
coh get_coram(instance_name, ...);
coh append_coram(coh coram, bool interleave, ...);

/*** Singleton control actions ***/
void coram_read(coh coram, void *offset,
           void *memaddr, int bytes);
tag coram_read_nb(coh coram, ...);
void coram_write(coh coram, void *offset,
            void *memaddr, int bytes);
tag coram_write_nb(coh coram, ...);
void coram_copy(coh src, coh dst, void *srcoffset,
           void *dstoffset, int bytes);
tag coram_copy_nb(coh src, coh dst, void *srcoffset,
           void *dstoffset, int bytes);
bool check_coram_done(coh coram, tag);

/*** Collective control actions ***/
collective_write(coh coram, void *offset,
                 void *memaddr, int bytes);
collective_read(coh coram, void *offset,
                 void *memaddr, int bytes);

/*** Channel control actions ***/
fifo_write(coh coram, Data din);
Data fifo_read(coh coram);

ioreg_write(coh coram, Data din);
Data ioreg_read(coh coram);
```

Figure 3: Control action definitions.

2.1 A Portable Memory Abstraction

Control actions. To facilitate a portable memory abstraction, control threads can only employ a predefined set of memory and communication primitives called control actions. Figure 3 illustrates an initial set of control actions defined in [4]. The various memory transfer control actions (e.g., *coram_write*) perform sequential accesses to and from the edge memory address space and can be blocking or non-blocking, depending on user preference. Non-blocking control actions return a tag that must be monitored using *check_coram_done*. The control actions also allow for CoRAM-to-CoRAM data transfers, which are useful for composing advanced memory structures (e.g., multi-level scratchpad or cache hierarchies). Figure 3 (bottom) describe control actions for communicating between a control thread and the application using asynchronous FIFOs and registers. As long as future FPGA-based computing devices preserve the existing set of control actions (and offer as much reconfigurable logic resources needed by the original application), application portability can be preserved without modification to the source code.

Advanced control actions. Beyond the simple control actions shown, a collective form of read and write control actions is also supported. In the collective form, *append_handle* is a helper function that can be used at compile-time to compose a static list of CoRAMs. The newly returned handle can then be used to perform transfers to the aggregated CoRAMs as a single logical unit. When operating upon the composed handle, sequential data arriving from memory can either be striped across the CoRAMs' local addresses in a concatenated or word-interleaved pattern. Such features allow the user to customize the partitioning of application data across the multiple distributed CoRAMs within the reconfigurable logic.

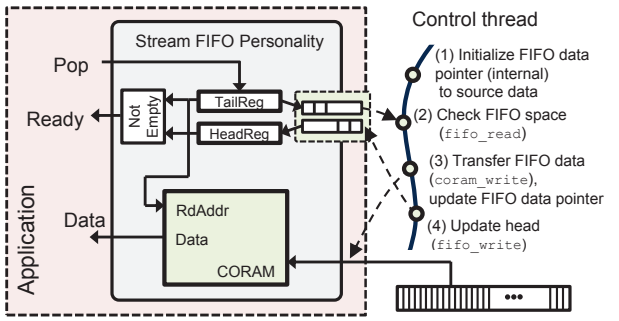

Figure 4: Input Stream FIFO Personality.

Based on the examples shown in Figure 3, it is not difficult to imagine that many variants of the above control actions could be added to the memory abstraction to support more sophisticated patterns (e.g., broadcast from one CoRAM to many). In a commercial production setting, control actions—like instructions in an ISA—must be carefully defined and preserved to achieve the value of portability and compatibility.

2.2 CoRAMs in Usage

An extended version of this paper [4] gives a detailed discussion of how CoRAMs and control threads were used to simplify the memory access mechanisms of various computational kernels such as Black-Scholes, Matrix Matrix Multiplication, and Sparse Matrix Vector Multiplication. In our case studies, CoRAMs and control actions were instantiated and wrapped within reconfigurable logic libraries to form portable, re-usable "memory personalities" crafted for a particular application's memory access pattern.

For example, to support the sequential memory accesses needed by the Black-Scholes kernel, a stream FIFO personality was developed in [4] that presents a simple FIFO interface from edge memory to the application. Figure 4 illustrates how the functionality of the stream FIFO personality is partitioned across reconfigurable logic, a single CoRAM, and an associated control thread. As Figure 4 shows, the control thread (1) initializes a memory pointer to the application's data, (2) queries the internal state of the FIFO to determine available occupancy (*fifo_read*), (3) performs a transfer of the data using a control action (*coram_write*), and (4) updates the stream FIFO's head register using a communication control action (*fifo_write*).

It is not difficult to imagine that other variants of the stream FIFO personality could be conceived, where the stream access pattern is not sequential but strided; or, the stream data itself could be sourced from another set of instantiated CoRAMs reserved as a backing on-chip storage. Such changes could be trivially made with few lines of code (e.g., replacing *coram_write* with *coram_copy*). Our paper in [4] also examined other memory personalities such as vector scratchpads and caches. It is conceived that in the future, soft libraries consisting of many types of memory personalities could be developed and re-used across many reconfigurable logic devices that support the CoRAM memory abstraction.

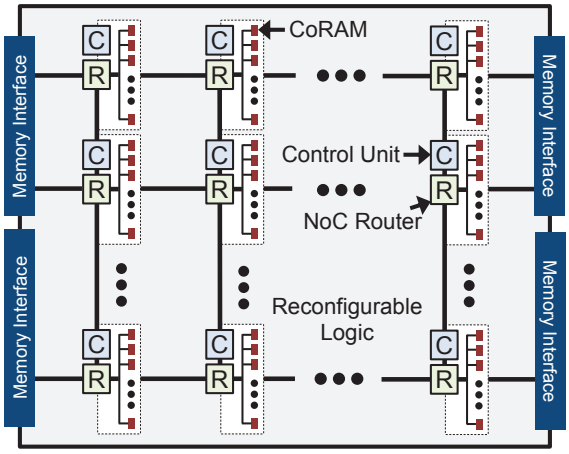

Figure 5: Conceptual Microarchitecture of Reconfigurable Logic with CoRAM Support.

3. IMPLEMENTATION METHODS

Like a general-purpose ISA, the CoRAM abstraction permits different microarchitectural implementations while maintaining a standard architectural view to applications. Ideally, a robust implementation of CoRAM should provide good memory subsystem performance across a wide variety of general applications without tuning or customization required by the user.

CoRAM Microarchitecture. Any implementation of the CoRAM abstraction naturally lends itself to three required mechanisms: (1) Control Units used to host the execution of control programs, (2) a Data Transport Engine to perform movement of data between CoRAMs and edge memory interfaces, and (3) the CoRAM storage resource itself used to present the data to the application. Figure 5 conceptually illustrates how the three required mechanisms could be implemented and physically arranged in a modern FPGA with CoRAM support.

CoRAMs, like embedded SRAMs in conventional FPGAs, are arranged into vertical columns. In Figure 5, the CoRAMs are further organized into discrete clusters attached to a Control Unit, which is a physical host responsible for executing the control programs that run within the cluster. A Control Unit can be realized by directly synthesizing a control program into a finite state machine within reconfigurable logic (e.g., using high-level synthesis flows) or can be implemented in hard logic as multithreaded microcontrollers that execute control programs directly. A control thread that manages more CoRAMs than available within a cluster could be replicated across multiple clusters.

The Data Transport Engine on the other hand is responsible for sending memory address requests on behalf of Control Units and delivering data responses from the edge memory interfaces (or from other CoRAMs). In cases where multiple CoRAMs are operated upon as logical units (e.g., *collective_write*), the Data Transport Engine must also break large data transfers into words and steer them accordingly to the constituent CoRAMs (based on the data partitioning a user desires). At the macroscale level, the Data Transport Engine is supported by a hierarchically-organized network-on-chip (NoC), where each NoC node is concentrated to service multiple aggregated CoRAMs (and to amortize the cost of each NoC router). The specific microarchitecture shown in Figure 5 illustrates each cluster serviced by a single network-on-chip router. The routers are laid out in a 2D mesh and provide global connectivity to and from the memory interfaces.

Soft versus hard logic. The most cost-effective approach to implementing CoRAM in the near term would be to layer all the required mechanisms on top of conventional FPGAs. In the long term, FPGA fabrics developed in mind with dedicated CoRAM architectural support can become more economical if certain features become popularly used. From the perspective of a fabric designed to support computing, we contend that a hardwired network-on-chip (NoC) offers significant advantages, especially if it reduces or eliminates the need for long-distance routing tracks in today's fabrics. Under the CoRAM architectural paradigm, global bulk communications are restricted to between CoRAM-to-CoRAM or CoRAM-to-edge. Such a usage model would be better served by the high performance (bandwidth and latency) and the reduced power and energy from a dedicated hardwired NoC that connects the CoRAMs and the edge memory interfaces. With a hardwired NoC, it is also more cost-effective (in area and energy cost) to over-provision performance to deliver robust performance across many different applications. Similarly, the control units used to host control threads could also support "hardened" control actions that are commonly used by developers.

3.1 Simulated Case Studies and Evaluation

The microarchitecture shown in Figure 5 was the subject of various case studies and a quantitative evaluation in [4]. To model a CoRAM-enabled FPGA, a detailed, cycle-level software simulator was developed using Bluespec System Verilog [2] to simulate the required CoRAM mechanisms. Pthreads were used to functionally host the execution of control threads and carefully throttled with synchronizations to model the notion of timing. For the three applications we examined in [4] (Black-Scholes, Matrix matrix multiplication, Sparse matrix-vector multiplication), the compute portions of the designs were written in synthesizable Bluespec and placed-and-routed on a Virtex-6 LX760 FPGA to determine the peak fabric processing throughput. Qualitatively, programming FPGA applications using the CoRAM abstraction substantially reduced the overall development efforts needed to bring functioning designs online. The results from our evaluation in [4] also showed that hardened, general-purpose CoRAM mechanisms can significantly outperform soft logic implementations for memory-intensive applications and that such mechanisms can be introduced in future reconfigurable logic designs with negligible overheads in area or power.

4. RELATED WORK

The CoRAM abstraction introduces the notion of memory management as an asynchronous control thread decoupled from the computation. The idea of separation of concerns for processing and memory is not new and has been examined extensively in the context of general-purpose processors [12, 6]. The MAP-200 machine, for example, employed an asynchronous integer address unit that was decoupled from the floating-point arithmetic pipeline [6].

An existing body of work has examined the design of soft memory hierarchies for FPGAs (e.g., [14, 7, 9, 10]). The most closely related work to CoRAM is the LEAP framework [1], which shares the objective of providing a standardized, platform-independent abstraction to FPGA programmers. LEAP abstracts away the details of memory management by exporting to the application a set of timing-insensitive, request-response interfaces to local client address spaces. CoRAM differs from LEAP by providing explicit user control over data movement between off-chip memory interfaces and the on-chip embedded SRAMs. The CoRAM abstraction could itself be used to facilitate the on-die cache mechanisms employed in an implementation of the LEAP memory abstraction.

5. CONCLUSIONS

Processing and memory are inseparable aspects of any real-world computing problems. A proper memory architecture is a critical requirement for FPGAs to succeed as a computing technology. This paper investigated a new memory architecture to provide deliberate support for memory accesses from within the fabric of future reconfigurable logic devices engineered for computing. This paper presented the CoRAM memory abstraction designed to match the requirements of highly concurrent, spatially distributed processing kernels that consume and produce memory data from within the fabric. In addition to improving performance and efficiency, the CoRAM architecture provides a virtualized memory environment as seen by the hardware kernels to simplify application development and to improve an application's portability and scalability.

6. REFERENCES

[1] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *FPGA'11: Proceedings of the 2011 ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, 2011.

[2] Bluespec, Inc. http://www.bluespec.com/products/bsc.htm.

[3] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *SASP'08: Proceedings of the 2008 Symposium on Application Specific Processors*, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[4] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Abstraction for FPGA-Based Computing. In *FPGA'11: Proceedings of the 2011 ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, 2011.

[5] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *MICRO-43: Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[6] E. Cohler and J. Storer. Functionally parallel architecture for array processors. *Computer*, 14:28–36, 1981.

[7] H. Devos, J. V. Campenhout, and D. Stroobandt. Building an Application-specific Memory Hierarchy on FPGA. *2nd HiPEAC Workshop on Reconfigurable Computing*, 2008.

[8] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2):69 –76, Feb. 2008.

[9] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang. FPGA Implementation of a Configurable Cache/Scratchpad Memory With Virtualized User-Level RDMA Capability. In *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, SAMOS'09, pages 149–156, Piscataway, NJ, USA, 2009. IEEE Press.

[10] P. Nalabalapu and R. Sass. Bandwidth Management with a Reconfigurable Data Cache. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IPDPS'05, pages 159.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[11] T. Ngai, J. Rose, and S. Wilton. An SRAM-programmable field-configurable memory. In *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*, pages 499 –502, May 1995.

[12] J. E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10:112–119, April 1982.

[13] Xilinx, Inc. Virtex-7 Series Overview, 2010.

[14] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *Proc. Field-Programmable Technology (FPT)*, pages 324–327, 2003.