Prototype and Evaluation of the CoRAM Memory Architecture for FPGA-Based Computing

Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, Ken Mai
Carnegie Mellon University, Computer Architecture Lab (CALCM)
coram@ece.cmu.edu

ABSTRACT

The CoRAM memory architecture for FPGA-based computing augments traditional reconfigurable fabric with a natural and effective way for applications to interact with off-chip memory and I/O. The two central tenets of the CoRAM memory architecture are (1) the deliberate separation of concerns between computation versus data marshalling and (2) the use of a multithreaded software abstraction to replace FSM-based memory control logic. To evaluate the viability of the CoRAM memory architecture, we developed a full RTL implementation of a CoRAM microarchitecture instance that can be synthesized for standard cells or emulated on FPGAs. The results of our evaluation show that a soft emulation of the CoRAM memory architecture on current FPGAs can be impractical for memory-intensive, large-scale applications due to the high performance and area penalties incurred by the soft mechanisms. The results further show that in an envisioned FPGA built with CoRAM in mind, the introduction of hard macro blocks for data distribution can mitigate these inefficiencies—allowing applications to take advantage of the CoRAM memory architecture for ease of programmability and portability while still enjoying performance and efficiency comparable to RTL-level application development on conventional FPGAs.

ACM Categories & Subject Descriptors
C.0 [Computer System Organization]: System Architectures
General Terms: Design, standardization
Keywords: FPGA computing, memory architecture

1. INTRODUCTION

In the quest for energy-efficient computing, Field Programmable Gate Arrays (FPGAs) have emerged as a class of general-purpose accelerators to address the increasing demands for performance while reducing energy consumption. Despite their raw capabilities, today's commodity FPGAs are impractical as general-purpose computing devices. When developing an application for an FPGA, designers are often confronted by: (1) low-level, error-prone hardware description languages (HDL), (2) "bare-bones" fabric with nothing but a sea of logic and I/O pins, and (3) low-level, vendor-specific interfaces and gateware that the application must be made compatible with.

CoRAM Memory Architecture. To address these limitations, the CoRAM memory architecture [1] is an endeavor to standardize and simplify how FPGA computing applications interact with memory and I/O, which is a critical step towards a portable FPGA abstraction. CoRAM presents a programmable, customizable view of memory that can be retargeted to different devices and platforms. The abstraction modifies the traditional FPGA's on-die SRAMs to act as in-fabric distributed portals to off-chip memory and I/O. A salient feature of CoRAM is the ability to program these customizable, on-die SRAMs using a software control thread that is portable and easy-to-use. Compared to the traditional approach where the FPGA memory hierarchy and I/O sub-system is hand-built at the RTL-level for each application, the CoRAM memory architecture can be used to efficiently support a broad range of applications.

Evaluating the Viability of CoRAM. The architectural features of CoRAM, however compelling, cannot be practical unless efficient underlying implementations are possible. In this paper, we investigate the extent to which CoRAM succeeds in serving as a performance- and cost-effective memory system replacement for FPGA-based applications. A critical aspect to be examined is whether the software-based control abstraction in CoRAM can adequately support FPGA applications with memory-intensive requirements. Our objectives require us to investigate the various ways in which the CoRAM memory architecture can be realized. At one end of the spectrum, CoRAM can be emulated on a conventional FPGA, albeit at the cost of soft logic area and performance. At the opposite end, general-purpose hard macro blocks can be embedded within conventional reconfigurable fabric to accelerate and to reduce the overhead of CoRAM operations. In this paper, we investigate both extremes and compare these implementations against traditional hand-built RTL designs on conventional FPGAs.

Prototyping Efforts and Results. Our investigation is supported by a full-featured prototype of a working CoRAM microarchitectural instance comprising: (1) a C-based language specification and compiler for software control threads, and (2) a highly parameterized RTL design of an optimized CoRAM microarchitecture retargetable to either standard cells (to model a future FPGA with hard

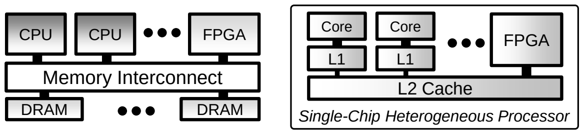

Figure 1: Assumed System Organizations.

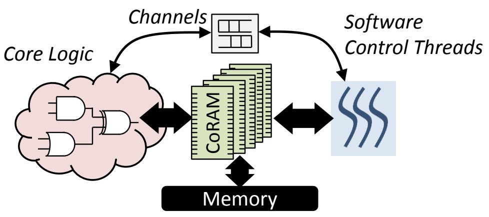

Figure 2: CoRAM Program Model.

CoRAM support) or conventional FPGAs (as a soft logic emulation). We select three diverse applications to evaluate the hard versus soft CoRAM implementations. Our evaluation results suggest that a soft emulation of CoRAM falls short in large-scale, memory-intensive applications due to the high performance and area penalties incurred by the soft mechanisms. However, our results show that the introduction of hard macro blocks for data distribution in a future FPGA with CoRAM support can mitigate these inefficiencies—allowing applications to achieve performance and efficiency comparable to tuned applications on a conventional FPGA.

2. CORAM BACKGROUND

Assumptions. At the system level, CoRAM assumes the existence of FPGA-based accelerators co-existing with general-purpose processors on a shared memory interconnect (see Figure 1). Within an FPGA, CoRAM assumes that one or more load-store interfaces provide external memory accesses at the boundaries of the fabric. The same assumptions are similarly applicable in a single-chip hetereogeneous multicore where cores and fabric co-exist on the same die.

CoRAM Program Model. When developing an application with CoRAM support, the user perceives a simplified view of fabric as depicted in Figure 2. The core logic component is an isolated, contiguous region of fabric that preserves the hardware-centric view familiar to designers that target FPGAs today. Applications that are mapped to core logic can be programmed with any hardware synthesis language supported by contemporary tools from low-level RTL (e.g., Verilog) to high-level languages (e.g., C-to-gates, Bluespec). To create a uniform abstraction that can be made portable across different devices and platforms, the CoRAM program model restricts all communication by the core logic to the external environment through the embedded CoRAM blocks shown at the edges of core logic in Figure 2. The embedded CoRAM blocks are user-instantiated, parameterizable SRAM modules that follow a similar usage paradigm of conventional embedded SRAMs [3]. On one hand, like SRAM blocks, CoRAM blocks offer customizable high-bandwidth storage, provide deterministic access ports to independent banks with local addresses, and can be composed with flexible aspect ratios to match the requirements of the application. On the other hand, unlike passive SRAM blocks, the contents of embedded CoRAM blocks are dynamically managed (such as loading and unloading against external main memory) using software control threads depicted in the right of Figure 2.

Software Control Threads. Software control threads form a fabric-distributed collection of logical, asynchronous control state machines for managing and mediating the data transfers between embedded CoRAM blocks and the edge memory interfaces. The software control threads and core logic are asynchronous peer entities in charge of data marshalling and computation, respectively; they communicate over bidirectional command queues. At a high level, the threads can be viewed as programmable mechanisms for prefetching an application's required data from the edge memory interface to the fabric-distributed embedded CoRAM blocks. At the lowest level, threads describe an ordered sequence of memory commands directed by control flow. The application developer relies solely on instantiated control threads to access shared memory and I/O from the beginning to end of computation.

Control threads can be used to express a rich variety of memory access patterns (e.g., random access, streaming, etc.) while maintaining portability. For example, a random-access cache controller could be implemented by combining soft logic and embedded CoRAM blocks (serving as the data array) and a control thread that implements a miss handler to memory. A stream FIFO could also be implemented by instantiating an embedded CoRAM block as a circular buffer with an associated control thread that fills or drains the buffer as needed. These portable memory building blocks can be expressed succinctly with relatively few lines of code (in most cases, under 100 lines of C) [1].

3. CORAM PROTOTYPE

There are two requirements to execute an application description in CoRAM: (1) interpreting or synthesizing high-level control threads into state machines, and (2) transporting data efficiently between memory interfaces and fabric-distributed embedded CoRAM blocks. To facilitate these, we developed a full-featured RTL prototype and control thread compiler for the CoRAM memory architecture. Our framework comprises: (1) the CoRAM Control Compiler (CORCC), which is an LLVM-based backend[1] that synthesizes C-based control thread programs into hardware finite state machines, (2) CONNECT [4], a flexible network-on-chip (NoC) generator tuned for the Virtex-6 architecture, and (3) pre-optimized macro blocks in Verilog for request handling, scoreboarding, and data distribution for the embedded CoRAM blocks. The RTL generated by our framework can be synthesized to standard cells for estimating the area, power, and performance in a hypothetical future FPGA with hardwired CoRAM support.

Figure 3 shows how a high-level CoRAM application is mapped into the synthesizable RTL generated by our framework. The embedded CoRAM blocks instantiated within the application are mapped into physical macro blocks called clusters, which aggregate up to 64 homogeneous 1024x32b SRAMs[2] into a single node. Embedded CoRAM blocks

[1]Low-Level Virtual Machine [2].
[2]Corresponding to the default aspect ratio of a typical FPGA BlockRAM [3].

	Max-Cfg Cluster		Mesh Router	
	Soft	Hard	Soft	Hard
LUTS+LUTRAMS	7615	-	6002	-
FFs	4741	-	1144	-
Clock (MHz)	108	840	125	610
SRAM Area (mm^2)	-	0.57	-	0.23
Die Area (mm^2)	23	0.74	18.1	0.3

Table 1: 65nm Characteristics of Single Cluster and Mesh Router.

	LUT	FF	BRAM	MHz
Microblaze (min-area-cfg)	1210	973	4	161
Stream Loop	155	118	0	345
Matrix Matrix Multiplication	2581	2802	0	192
Non-blocking Cache Miss Handler	242	316	0	354
Stream FIFO Producer Thread	544	523	0	204

Table 2: Control Thread Synthesis (Virtex-6, -2).

with aspect ratios larger than 1024x32b are constructed by spanning across multiple SRAMs within a cluster.[3] The clusters are connected to external memory links through a CONNECT-generated NoC. Figure 3 illustrates how each of the clusters are attached to one or more Control Finite State Machines (C-FSM) generated by CORCC from the control threads. At run-time, the C-FSMs issue memory commands to the NoC—the subsequent memory responses are collected at the clusters, which are responsible for steering and aligning the memory data to the destination CoRAM blocks.

Soft vs. Hard Macro Blocks. The C-FSMs, clusters, and NoC are performance- and area-critical macro blocks that can either be soft-emulated in today's FPGAs or embedded as hard logic in a future FPGA. Although we omit the details, a substantial effort was invested in optimizing the macro blocks for the Virtex-6 architecture. Table 1 compares the cost of a single maximally-configured cluster and mesh router for both hard and soft implementations.[4] Note that a maximally-configured cluster supports up to 64 SRAMs, 16 control threads, and 128 concurrent transactions to memory. In practice, a soft cluster can be configured less aggressively depending on the application—as little as 2KLUTs. When normalized to die area, both the hard cluster and mesh router achieve an order-of-magnitude improvement in area efficiency as expected. The hard macro blocks also operate at higher clock frequencies (e.g., 610MHz vs. 125MHz for the mesh router). Table 2 further shows the soft logic area of various control threads synthesized to FPGA fabric using CORCC. Across most applications, the area consumption is modest when compared to a minimally-configured Microblaze core, which suggests that control threads can be supported practically in conventional fabrics.

4. EVALUATION

Methodology. For our evaluation, we study the use of CoRAM in three diverse RTL applications reflecting bandwidth-bound, compute-bound, and latency-bound characteristics [1]: (1) Black-Scholes Options Pricing (BS), (2) Matrix-Matrix Multiplication (MMM), and (3) Sparse Matrix-Vector Multiplication (SpMV). Our evaluation considers the cost and performance of the CoRAM memory architecture across multiple dimensions: (1) VLSI technology, (2) NoC topology, and (3) hard vs. soft logic implementations. Table 3 shows the parameters and characteristics of the selected FPGA configurations. All configurations (with or without hard CoRAM support) assume the existence of hard memory controllers that provide high external memory bandwidth to the fabric. All soft-logic designs are tested against different NoC topologies (ring, mesh, crossbar) to reflect the option of customization in a soft-logic implementation. Furthermore, in the soft designs, the number of clusters and SRAMs per cluster are tuned for each respective application. The hard implementation assumes a mesh topology with a fixed number of clusters and SRAMs per cluster.

Results. Figure 4 (top) shows the performance trends for all applications. On the x-axis, labels prepended with "soft" indicate designs using a soft-logic implementation on a conventional FPGA. Designs labeled "hard" indicate FPGAs with dedicated CoRAM support operating at 4X the clock rate of the soft-logic design. The right-most design point of each graph shows the results for an "ideal" application running on a conventional FPGA that can access external DRAM with no on-chip delay or soft logic area overheads. Figure 4 (bottom) shows the LUT area breakdown and LUT-

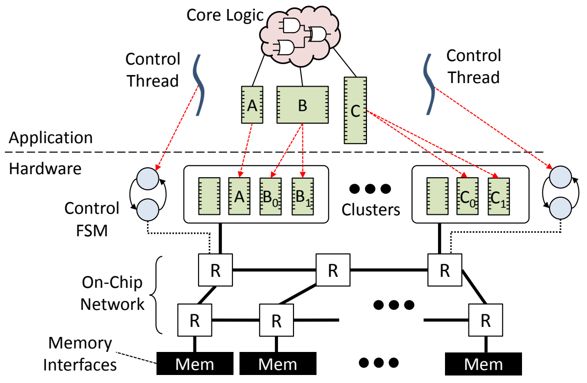

Figure 3: CoRAM Microarchitecture.

		500K	1M	2M	4M
	Technology	45nm	32nm	22nm	16nm
	Die Area (mm^2)	600	600	600	600
	LUTs (K)	500	1000	2000	4000
	Frequency (MHz)	200	200	225	250
	Bandwidth (GB/s)	25.6	51.2	102.4	204.8
	4kB CoRAM blocks	1024	2048	4096	8192
Soft	Cluster/NoC clock (GHz)	0.2	0.2	0.225	0.25
	Cluster/NoC link width	128	128	128	128
	Clusters/Nodes	Application-dependent			
Hard	Cluster/NoC clock (GHz)	0.8	0.8	0.9	1
	Cluster/NoC link width	128	128	128	128
	Clusters	16	32	64	128
	Nodes (clusters+DRAM links)	20	40	80	160
	SRAMs per Cluster	64	64	64	64
	Die Area Ovhd (%)	1.7	1.7	1.4	1.4

Table 3: Evaluation Parameters.

[3]The maximum size of the cluster can either be set automatically or configured by the user.
[4]FPGA results were obtained with XST 13.1 for the Virtex-6; ASIC results were obtained using CACTI 6.5 [5] and Synopsys Design Compiler configured with commercial 65nm standard cells.

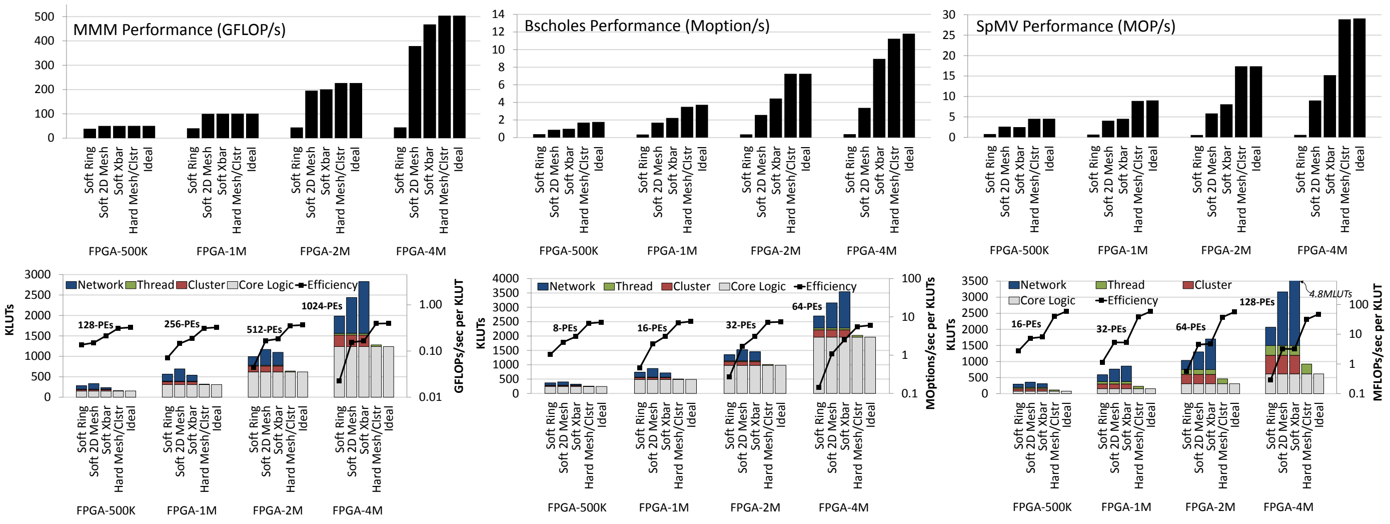

Figure 4: Performance and Area Efficiency Trends.

normalized performance for the same corresponding performance points.

What Is The Gap Between Soft versus Hard? The overall trends in Figure 4 show a gap in performance and efficiency between soft and hard implementations of CoRAM. In MMM, the performance is comparable across all design points (except for the soft ring network, which suffers from high contention). When normalized to area, however, a significant gap in efficiency (about 2X) separates soft versus hard implementations of CoRAM. The NoCs expend considerable soft logic area in buffering, which is the largest contributer to overhead. In the more memory-intensive applications such as BS and SpMV, the biggest impact on performance is the increased queuing delay as a result of higher latency and contention in the soft data distribution mechanisms. This impacts SpMV the greatest (which is latency-sensitive)—resulting in a 2X gap in performance between the soft versus hard CoRAM implementations.

Do Software Control Threads Limit Efficiency or Performance? A notable result of Figure 4 (bottom) is that the synthesized control threads constitute only a relatively small fraction of the overall area. What also stands out is that with an efficient implementation of CoRAM using hard macro blocks, the use of software-based control threads does not limit the peak performance potential of the various applications. Both SpMV and Black-Scholes, for example, were able to achieve bandwidth-limited performance even though the logic used to generate their memory accesses were described using high-level, C-based control threads. These results support the hypothesis that a high-level software-based abstraction for memory management does not fundamentally limit the performance potential of memory-intensive FPGA-based applications.

Hard CoRAM vs. Conventional FPGA. A key result of Figure 4 is that applications mapped to a hard implementation of CoRAM are capable of achieving performance and efficiency comparable to tuned applications on conventional FPGAs (labeled "ideal"). Recall, the measured results of "ideal" applications are based on simulations and synthesis of core logic that do not incur any overheads in latency or area from memory control logic or data distribution between the core logic and the external memory interfaces. This suggests that hard macro blocks for CoRAM can reduce the tuning effort needed by designers when optimizing the memory accesses for an application.

5. CONCLUSIONS

This paper presented a full-featured RTL prototype of the CoRAM memory architecture for FPGA-based computing. Our prototype enabled us to investigate designs across the continuum—from soft-logic emulation on conventional FPGAs to hard macro blocks in future FPGA fabrics. Despite our best efforts, our soft implementations of CoRAM fell short of being practical in large-scale, memory-intensive applications. Our results show that the hardening of macro blocks for data distribution in CoRAM enables application development using a high-level software memory abstraction to achieve performance and efficiency that is comparable to optimized RTL development on a conventional FPGA.

6. ACKNOWLEDGEMENTS

Funding for this work was provided in part by NSF CCF-1012851 and by Altera. We thank the anonymous reviewers, members of CALCM, and Chuck Thacker for their comments and feedback. We thank Xilinx and Altera for their FPGA and tool donations. We thank Bluespec for their tool donations and support.

7. REFERENCES

[1] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *Proceedings of FPGA'11*.

[2] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'04*.

[3] T. Ngai, J. Rose, and S. Wilton. An SRAM-programmable field-configurable memory. In *Proceedings of CICC'95*.

[4] M. K. Papamichael and J. C. Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *Proceedings of FPGA'12*.

[5] D. T. S. Thoziyoor, D. Tarjan, and S. Thoziyoor. Cacti 4.0. Technical Report HPL-2006-86, HP Labs, 2006.