2013 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines

# ShrinkWrap: Compiler-Enabled Optimization and Customization of Soft Memory Interconnects*

Eric S. Chung
*Microsoft Research*

Michael K. Papamichael
*Carnegie Mellon University*

***Abstract*—Today's FPGAs lack dedicated on-chip memory interconnects, requiring users to (1) rely on inefficient, general-purpose solutions, or (2) tediously create an application-specific memory interconnect for each target platform. The CoRAM architecture, which offers a general-purpose abstraction for FPGA memory management, encodes high-level application information that can be exploited to generate customized soft memory interconnects. This paper describes the ShrinkWrap Compiler, which analyzes a CoRAM application for its connectivity and bandwidth requirements, enabling synthesis of highly-tuned area-efficient soft memory interconnects.**

## I. Introduction

The lack of a dedicated on-chip memory interconnect continues to be a weak link when it comes to creating a balanced, high-performance design on an FPGA. Today's FPGAs do not provide native on-chip interconnects such as a memory bus or network-on-chip (NoC) and continue to rely on the classic reconfigurable routing architecture for wire-like connections between many distributed clients and off-chip memory interfaces. When building the memory system for an application, designers typically face two opposing yet equally undesirable choices: (1) rely on inefficient, vendor-specific buses and NoCs realized on top of reconfigurable logic, or (2) by hand, manually create and tune a memory interconnect matched to the application's latency and bandwidth requirements.

In this paper, we present a new compiler called ShrinkWrap that enables graceful optimization and tuning of the memory interconnect on a per-application basis. Unlike conventional flows for targeting "bare-metal" FPGA abstractions, ShrinkWrap is fundamentally layered upon the recently proposed CoRAM FPGA computing abstraction [1]. The use of CoRAM uniquely exposes both flexibility and high-level information that can be exploited during the interconnect generation process.

## II. What Is CoRAM?

CoRAM was originally conceived as a portable, general-purpose architecture layer for FPGA-based computing [1]. The CoRAM programming model offers (1) an easy-to-use software abstraction for FPGA on-die memory management and (2) portability across any supported FPGA platform, either through native support in future FPGAs or based on soft-logic implementations on existing FPGAs. As illustrated in Figure 1 (top), CoRAM deliberately partitions an application into two separate components: Core Logic and software control threads. Core Logic, typically comprised of application-specific cores on the FPGA, are attached to distributed CoRAM blocks (SRAMs) with logical connections to any off-chip DRAM interface (seen by the user as a global shared memory with a single address space). To perform off-chip memory accesses, a collection of C-based software control threads are used to specify the sequencing and movement of data between external DRAM and the on-chip CoRAMs.

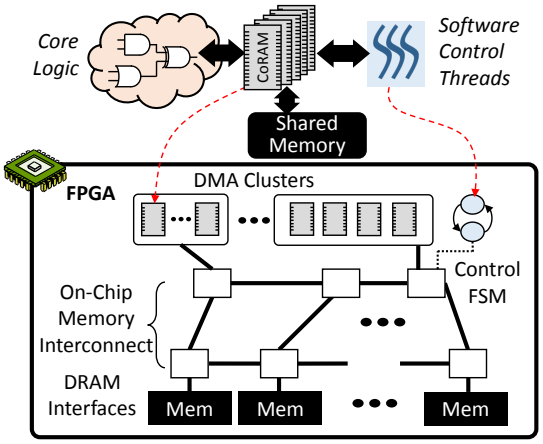


Figure 1. The CoRAM Architecture (adapted from [2]).

**Compilation Flow.** A C-based software control thread captures an applications' high-level memory access behavior but does not specify an actual implementation (in similar spirit to a portable ISA). In practice, a tool chain must translate the high-level threads and their memory access invocations onto a memory system that can service and track logical memory accesses and efficiently move data between edge DRAM interfaces and the distributed CoRAM blocks.

The bulk of these mechanisms—highlighted in the proposed microarchitecture shown in Figure 1 (bottom)—comprise a scalable on-chip memory interconnect and multiple DMA clusters used to provide connectivity between up to thousands of distributed CoRAM blocks and a multitude of DRAM interfaces. To reduce the number of end-points on the network, multiple CoRAM blocks are grouped into clusters, which supply a fixed allocation of off-chip memory bandwidth. Control threads in the programming abstraction can be further implemented by direct synthesis onto finite state machines or executed on dedicated micro-controllers.

*A full-length technical report is available at www.ece.cmu.edu/~coram.

978-0-7695-4969-9/13 $26.00 © 2013 IEEE
DOI 10.1109/FCCM.2013.56

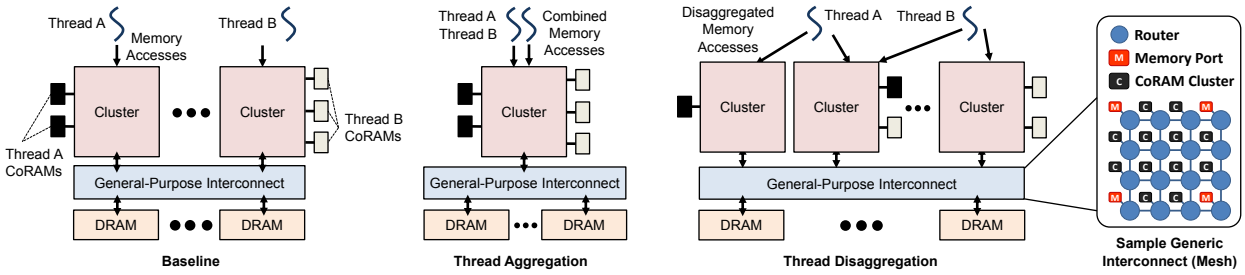


Figure 2. Simple Mapping Strategies from Threads to Memory Interconnect.

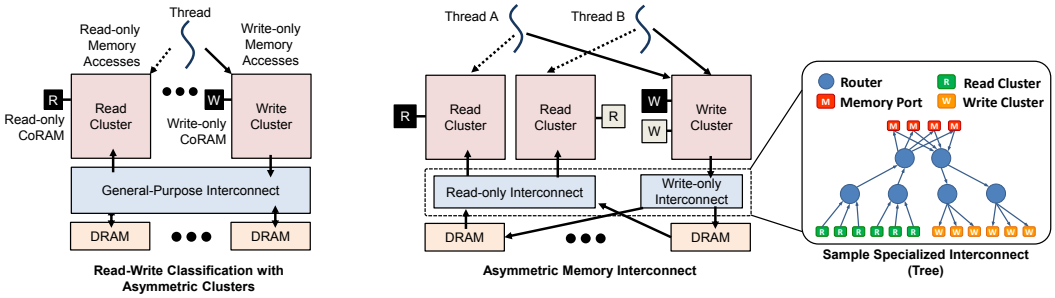


Figure 3. Advanced Mapping Strategies from Threads to Memory Interconnect.

## III. The ShrinkWrap Compiler

Control threads in CoRAM appear as high-level specifications for memory accesses but encode information such as: (1) actual connectivity between specific CoRAMs and memory ports in the system, (2) the actual direction of memory traffic, and (3) with static analysis, the actual amount of traffic demand. With this knowledge, ShrinkWrap introduces several compile-time optimizations to guide the soft memory interconnect generation process, enumerated in detail below.

**Baseline: Black-Box Mapping Strategy.** Figure 2 (left) shows a baseline, black-box mapping strategy used in prior compilation approaches [2]. In this approach, the control threads and their associated CoRAM blocks are each assigned to individual private clusters. At the macro scale, a "black-box" general-purpose interconnect with a fixed topology and capacity (i.e., number of nodes, link width, frequency) is first selected by the user to provide bulk data transfers between the clusters and the off-chip DRAM interfaces. The clusters and DRAM interfaces are then assigned and mapped to available end-points on the network. Although this is the simplest strategy to implement, it is conservatively over-provisioned in bandwidth and could introduce significant overheads in area.

**Thread (Dis-)Aggregation.** Thread Aggregation and Disaggregation improves upon the baseline method by statically analyzing the control threads (and the CoRAMs they manage) and combining (or decomposing) their assignment to cluster resources. Figure 2 (middle) illustrates the thread aggregation method, in which CoRAMs belonging to two threads are assigned to a single cluster. Thread aggregation reduces soft logic area consumption by decoupling the number of threads and their resources (i.e., number of logical network end-points) from the physical number of interconnect end-point switches. Aggregation sacrifices memory bandwidth in exchange for a lower area cost and is most suitable for compute-intensive applications with high arithmetic intensity. Thread disaggregation, on the other hand, spreads the CoRAMs managed by a logical thread across multiple clusters. Instantiating additional clusters can increase the overall throughput of the memory system at the cost of additional endpoints on the interconnect. Disaggregation is most suitable for memory-intensive applications with high bandwidth and low arithmetic intensity.

**Read-Write Classification with Asymmetric Clusters.** A more advanced form of thread disaggregation is illustrated in Figure 3 (left), in which CoRAM blocks belonging to a single thread are classified and mapped separately to clusters that are specialized for unidirectional traffic. In this setting, the ShrinkWrap compiler statically analyzes the C-based software control thread program and classifies the CoRAMs based on their usage over the lifetime of application execution: read-only, write-only, and read-write. A read-only CoRAM, for example, is only used by the application to perform in-bound data transfers from the network/DRAM. Once classified, CoRAMs of each type are mapped to clusters that only support one-way communication to the memory interconnect. A read cluster (Figure 3, middle), consumes less area than a standard cluster and only provides one-way scatter-gather capability and a single inbound link from the interconnect. (Bidirectional read-write CoRAM clusters are still attached to both memory interconnects.)

**Asymmetric Memory Interconnect.** Figure 3 (right) illustrates all of the previous optimizations combined, where

| Topology | Router | Flow Ctrl | LUTs | MHz |
|---|---|---|---|---|
| 4x4 Mesh (baseline) | VC-based | credits | 46361 | 112 |
| 4x4 Mesh | VOQ-based | peek | 31197 | 126 |
| Symmetric Tree | VOQ-based | peek | 5137 | 142 |
| Asymmetric Tree | VOQ-based | peek | 3642 | 220 |
| All networks have 128-bit links and equal buffering per router. | | | | |

Table I
SYNTHESIS RESULTS FOR INTERCONNECT ALTERNATIVES.

logically centralized threads and CoRAMs are now entirely decoupled from physical clusters and physical end-points on the memory interconnect. In the final optimization, the read, write, and read-write clusters can further be mapped into asymmetric, decoupled read- and write-only memory interconnects, which can now be configured and optimized independently in network topology and capacity for each traffic direction. In the ShrinkWrap compiler, the user has the option of enabling a subset or all of the optimizations.

## IV. INTERCONNECT OPTIMIZATIONS

Prior compilation approaches for CoRAM relied on the public-domain CONNECT framework [3] to generate black-box memory interconnects—i.e., the topology, router architecture and other details, such as in-order delivery guarantees and virtual channel support, were hidden from the application. ShrinkWrap utilizes a new version of the CONNECT NoC generation framework that includes multiple new optimizations and features, including support for building asymmetric memory interconnects. Unlike standard meshes or rings used in previous studies [2], ShrinkWrap leverages parameterizable, unidirectional, tree-based topologies that are more efficient at handling cluster-to-memory traffic.

ShrinkWrap also incorporates router-level optimizations that were not available in the original CONNECT framework: (1) the elimination of Virtual Channels (VCs) due to the use of independent networks for read and write traffic, which vastly simplifies the router design and lowers its cost, (2) an efficient way to implement virtual output queueing (VOQ) by packing multiple queues into a single physical LUTRAM, and (3) the use of peek-based flow control (as opposed to more expensive credit-based), which allows a router to directly query the occupancy of adjacent routers.

Table I shows FPGA synthesis results assuming a soft logic CoRAM system with 16 endpoints (4 memory ports and 12 CoRAM clusters), starting from a baseline 4x4 VC-based mesh NoC that employs credit-based flow control and progressing all the way to an optimized asymmetric unidirectional VOQ based pruned tree topology that employs peek flow control. Our results show significant area and frequency improvements—from 46K to 3.6K LUTs and 112MHz to 220MHz.

## V. EVALUATION

We compare generated RTL designs that employ ShrinkWrap optimizations versus the conventional black-box method. We developed two FPGA accelerators using the CoRAM API [1]: (1) Dense, Single-Precision Matrix Multiply (DMM), and (2) Single-Precision Black-Scholes Option Pricing (BS). Our study targets a large-scale Xilinx Virtex-6 LX760 FPGA [4] allowing up to 760 BlockRAMs to be used as CoRAMs. We model in RTL an aggressive memory system with 25.6 GB/sec peak bandwidth exposed as 16 memory ports at 100MHz. All performance results are collected by simulating the generated designs at 100MHz using Synopsys VCS. All area and timing results are reported using Xilinx ISE 13.1 targeting the LX760 (-2).

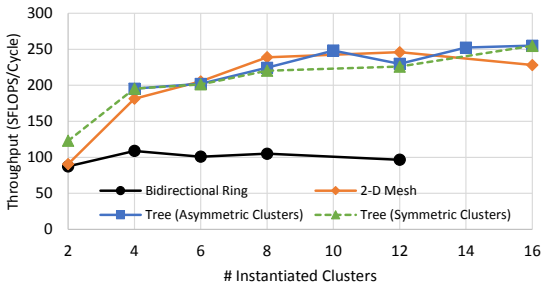


Figure 4. Dense Matrix Multiply Throughput.

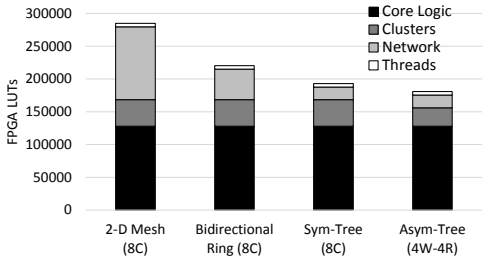


Figure 5. Dense Matrix Multiply Area Comparison.

| Throughput Efficiency (FLOPs/Cycle/LUTs x 1e4) | |
|---|---|
| 2-D Mesh (6C) | 8.4 |
| Bidirectional Ring (8C) | 4.75 |
| Sym-Tree (8C) | 11.4 |
| Asym-Tree (4W-4R) | 12.4 |

Table II
DENSE MATRIX MULTIPLY THROUGHPUT/AREA EFFICIENCY.

### A. Methodology

Using the CoRAM API, experimenting with different soft memory interconnect configurations can be achieved without any changes to the control thread program or the core logic of the application. For BS and DMM, we configure ShrinkWrap to emit conventional networks (e.g., 2-D mesh) and both symmetric and asymmetric tree-based networks and clusters. The ability to adjust the number of clusters of each type (read, write, read-write) is a "major knob" to scale the memory system bandwidth in proportion to traffic demand. For a fixed number of clusters, ShrinkWrap uniformly maps instantiated CoRAMs across available resources.

**Optimizing Conventional Interconnects.** In the conventional interconnects, the clusters are connected by a point-to-point network such as a 2-D mesh or ring. For tuning, we systematically instantiate a few clusters and gradually scale them (along with the network size) until performance

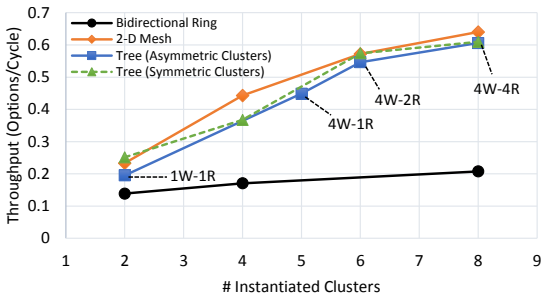


Figure 6. Black-Scholes Throughput.

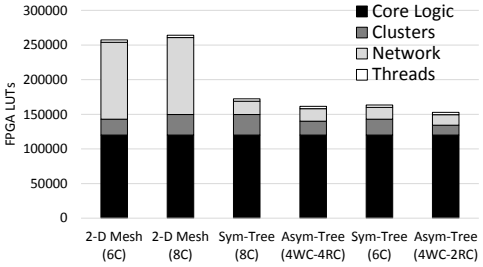


Figure 7. Black-Scholes Area Breakdown Comparison.

saturation is reached (or resources are exhausted). This performance is established as a baseline for all subsequent experiments. In the case of a bidirectional ring, memory interfaces and clusters are interspersed to reduce hotspots. In the 2-D mesh, memory interfaces are placed near the edges, while clusters occupy remaining nodes.

**Optimizing Tree-based Interconnects.** With optimizations enabled, ShrinkWrap replaces the conventional networks with tree-based networks. In a symmetric tree-based network, identically-configured tree networks are used to facilitate memory traffic in both directions between clients and memory. Asymmetric tree-based networks allow the independent trees to be customized and scaled separately. We also have the option to select specialized clusters—referred to as RW-Cluster (full read-write support), W-Cluster (write-to-CoRAM only), and R-Cluster (read-from-CoRAM only). To find the optimal enumeration of cluster types, we explore different configurations by starting with an optimal number of RW-Clusters, then systematically converting them to asymmetric R- and W-Clusters.

### *B. Results*

**Dense Matrix Multiply Discussion.** Figure 4 shows the throughput of DMM as a function of instantiated clusters. The "Tree (Symmetric Clusters)" shows the isolated performance of the tree using RW-Clusters only. The curve labeled "Tree (Asymmetric Clusters)" uses a fixed ratio of R-Cluster and RW-Clusters to match the bandwidth demands of DMM—e.g., at 8 clusters, the "Tree (Asymmetric Clusters)" instantiates 4 W-Clusters and 4 RW-Clusters. Across all networks except the ring, throughput scales with the total number of clusters until performance is saturated (at about 8 clusters).

| Throughput Efficiency (FLOPs/Cycle/LUTs x 1e6) | |
|---|---|
| 2-D Mesh (6C) | 2.2 |
| 2-D Mesh (8C) | 2.4 |
| Sym-Tree (8C) | 3.0 |
| Asym-Tree (4W-4R) | 3.3 |
| Sym-Tree (6C) | 3.1 |
| Asym-Tree (4W-2R) | 3.3 |

Table III
BLACK-SCHOLES THROUGHPUT/AREA EFFICIENCY.

**Area Comparison.** Although "Asymmetric Tree" and "2-D mesh" achieve comparable performance, Figure 5 shows that the 2-D mesh network incurs significantly more area overhead. The ShrinkWrap optimizations that employ tree-based networks prune out unnecessary links and datapaths—reducing the overhead by more than 2X. The classification of traffic (i.e., splitting CoRAMs into separate W-, R-, and RW-Clusters) further improves the cluster area, as illustrated in Figure 5, which compares the tree with and without symmetric clusters ("Sym-Tree" vs. "Asym-Tree"). Table II shows the throughput of DMM normalized to area. The asymmetric approach provides a 37% increase in overall efficiency. In general, the best approach combines the efficient tree-based networks with asymmetric clusters configured and tuned to the application's requirements.

**Black-Scholes Discussion.** The trends observed in the DMM kernel are similar in BS. The optimal ratio of R- to W-Clusters closely tracks the intrinsic Black-Scholes memory bandwidth requirement (5 Reads and 2 Writes per clock cycle). As seen in Figures 6 and 7, the most efficient ratio for the asymmetric tree is 4W-2R (while 4W-4R only offers incrementally improved performance). Table III shows that BS paired with an asymmetric tree with 4 W-Clusters and 4 R-Clusters achieves 48% improvement in area efficiency relative to the 2-D mesh.

## VI. CONCLUSION

General-purpose soft memory interconnects on FPGAs are convenient but untailored to the specific needs of a given application. The ShrinkWrap compiler, in conjunction with the CoRAM program abstraction, enables automatic synthesis of application-specific networks that exploit information preserved by CoRAM control threads. For two non-trivial applications, ShrinkWrap achieves significant increases in area efficiency (up to 48%) without requiring changes to the application and without significant impact on performance.

## REFERENCES

[1] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing. In *FPGA*, 2011.
[2] 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 *FPGA*, 2012.
[3] M. K. Papamichael and J. C. Hoe. CONNECT: Re-Examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. In *FPGA*, 2012.
[4] Xilinx, Inc. Virtex-7 Series Overview, 2010.