Automatic Construction of Program-Optimized FPGA Memory Networks

Hsin-Jung Yang
Massachusetts Institute of Technology, CSAIL
hjyang@csail.mit.edu

Kermin Fleming
Intel Corporation
SSG Group
kermin.fleming@intel.com

Felix Winterstein
Imperial College London
CAS Group
f.winterstein12@imperial.ac.uk

Annie I. Chen
Massachusetts Institute of Technology, EECS
anniecia@mit.edu

Michael Adler
Intel Corporation
SSG Group
michael.adler@intel.com

Joel Emer
Massachusetts Institute of Technology, CSAIL
emer@csail.mit.edu

ABSTRACT

Memory systems play a key role in the performance of FPGA applications. As FPGA deployments move towards design entry points that are more serial, memory latency has become a serious design consideration. For these applications, memory network optimization is essential in improving performance. In this paper, we examine the automatic, program-optimized construction of low-latency memory networks. We design a feedback-driven network compiler, which constructs an optimized memory network based on the target program's memory access behavior measured via a newly designed network profiler. In our test applications, the compiler-optimized networks provide a 45% performance gain on average over baseline memory networks by minimizing the impact of network latency on program performance.

1. INTRODUCTION

FPGA-based accelerators have great potential to achieve better performance and energy-efficiency compared to general-purpose solutions because the FPGA permits the tailoring of hardware to a particular application. However, as FPGAs and FPGA-based systems have grown, traditional approaches such as low-level hardware development and system-level hand-tuning have strained in the face of design complexity. To address FPGA programmability challenges, recent work has focused on raising the level of design abstraction by providing high-level programming models [1][2][3][4] as well as offering optimized and reusable service implementations [5][6][7]. However, high-level abstractions and productivity sometimes come at the expense of intelligent control and performance, resulting in a performance gap between a generated system and a manually optimized design. To construct high-performance designs while maintaining high productivity, it is essential to have a compiler that automatically optimizes the abstract service implementations in an application-specific manner on behalf of programmers.

In this work, we focus on FPGA memory systems, the performance of which is critical to the overall program performance for a broad class of applications. Unlike general purpose processors, where the memory system is fixed at design time, FPGAs offer the opportunity to intelligently customize the complete memory system; for example, the number, properties, and topology of cache hierarchies can be selected based on the behavior of a particular program. Furthermore, a specific optimization on FPGA does not need to provide a large average benefit (as is required in processors) because the optimization can be applied only when it can benefit the target application, avoiding unnecessary overhead.

Previous work on multi-level FPGA memory hierarchies has generally focused on the microarchitecture and optimization of on-chip caches to achieve higher cache bandwidth and hit rate [8][9][10]. However, with the rise of serial design entry points for FPGAs, such as C-based kernels compiled through high-level synthesis (HLS), memory latency has become a first-class design consideration. Though parallel, HLS programs are sometimes less parallel than conventional designs written at register transfer level (RTL), making them more sensitive to latency in the memory subsystem. In this work, we focus on the construction of low-latency memory networks. We aim to improve program performance by customizing networks connecting different levels of caches in the memory hierarchy for each target application. Memory network customization is especially valuable when the memory clients of the target program have asymmetric memory access behavior. For example, a memory client that is more latency-sensitive, possibly due to lower data locality or lower request-level parallelism, should be granted a faster network path. Constructing a program-optimized cache network requires the evaluation of cost-performance tradeoffs on a per-application basis. Since the design space of network topologies is quite large, manual exploration is unattractive, and an automated solution is desirable.

In order to automate the design space exploration, we first propose a new communication abstraction for centralized services to separate the functionality of the service network from physical topologies, allowing compilers to optimize the memory network under the proposed abstraction without changing other components in the memory system. To construct a program-optimized network implementation, we need a systematic way of evaluating the performance impact of different network configurations and characterizing the memory access behavior of the program. We therefore introduce a dynamically-configurable network profiler, which can be used to emulate different network topologies for the target application without reconstructing the hardware. In the network profiler, program instrumentation logic is inserted at each memory client in order to measure the client's latency and bandwidth demands.

FPGA '17, February 22-24, 2017, Monterey, CA, USA
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02. . . $15.00
DOI: http://dx.doi.org/10.1145/3020078.3021748

```
interface MEM_IFC#(type t_ADDR, type t_DATA);
    method void readRequest(t_ADDR addr);
    method t_DATA readResponse();
    method void write(t_ADDR addr, t_DATA data);
endinterface
```

Figure 1: LEAP memory interface

Armed with an abstraction and a means of program introspection, the final step is to develop algorithms for selecting an optimal network topology. We present an integer linear programming (ILP) formulation to determine an optimized tree-based network that minimizes the network latency impact on program performance. We also propose an efficient approximation algorithm that solves this optimization problem in polynomial time using dynamic programming (DP). To implement the optimized physical network, we extend the LEAP Memory Compiler (LMC) proposed in [11]. The compiler takes the profiling results obtained from the network profiler and uses the above optimization techniques to automatically construct an optimized network for the target application.

To test the scalability and robustness of our algorithms, we also consider an emerging class of FPGA workloads: multi-program applications, which we view as representative of future FPGA deployments, especially in the data center context. To support the needs of such deployments and to help amortize large FPGAs, FPGA virtualization has been proposed [12], allowing several user programs to be simultaneously mapped to the same FPGA. On a virtualized FPGA platform, it is common to have a large number of memory clients sharing memory system resources. In order to balance the performance across competing applications, we introduce some quality-of-service controls into the compiler-generated memory network to control fairness among multiple programs.

We evaluate the performance of our automatically-generated cache networks on both single-program and multi-program applications. The single-program applications we target contain HLS-compiled computational kernels that are sensitive to memory latency. For these applications, on average, the program-optimized tree networks achieve a 45% performance gain over the baseline memory networks and a 17% performance gain over the partitioned ring-based networks constructed by the original LMC. For multi-program applications, the tree network with bandwidth control also achieves better fairness by preventing throughput-oriented applications from saturating the memory system bandwidth.

2. BACKGROUND

Our exploration of program-optimized memory network construction builds upon prior work in the automatic synthesis of FPGA memory subsystems: LEAP memories [13][14] and the LEAP Memory Compiler (LMC) [11].

LEAP private memories [13] provide FPGA programs a general, in-fabric memory abstraction with a simple read-request, read-response, write interface as shown in Figure 1. Programmers can instantiate as many memories as needed to store arbitrary data types, and each instantiated memory represents a logically private address space. LEAP memories also support arbitrary address space sizes, which may be larger than the total capacity of FPGA physical memories. To provide the illusion of large address spaces, LEAP exploits the host virtual memory as a backing store and uses FPGA physical memories, including both on-chip and on-board memories, as caches to maintain high performance. LEAP coherent memories [14] extend the private memory abstraction to maintain coherency and consistency of accesses to a shared memory space. A program may declare multiple, independent coherent address spaces.

Similar to the load-store interface of memory systems on general-purpose machines, the abstract interfaces of LEAP memories do not

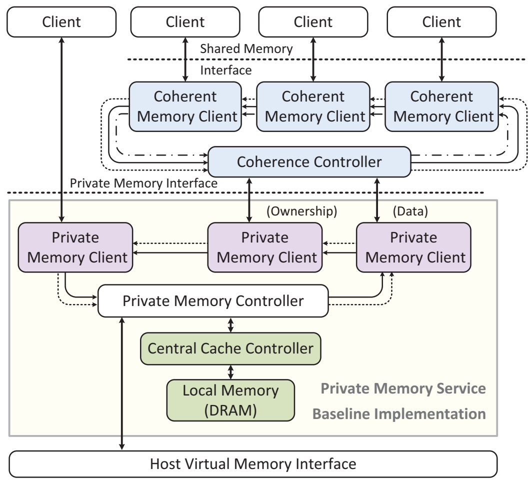


Figure 2: An example of LEAP memory hierarchy

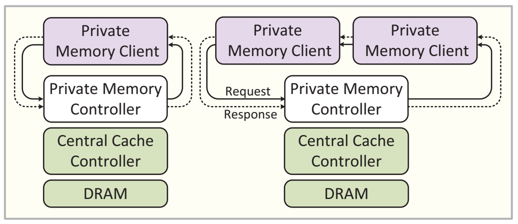


Figure 3: A partitioned ring-based memory network created by LMC

imply any implementation details of the underlying memory system, such as how many levels of cache are in the memory hierarchy or the topology of cache networks. This ambiguity provides compilers significant freedom to construct memory hierarchies based on properties of the target application and the target platform.

Figure 2 shows an example of a typical LEAP memory hierarchy which integrates one private memory and three coherent memories instantiated in the user program. LEAP coherent memories are built on top of the private memory hierarchy: the coherence controller of each shared memory space uses two private memories as data and coherence ownership stores. In each memory client, a local cache can be optionally constructed using on-chip SRAMs. As a baseline, all private memory clients are connected to a single, centralized controller hierarchy, which manages accesses to a central cache implemented with aggregated FPGA board-level memories. Within the central cache, each private and shared memory space is uniquely tagged, enforcing a physical separation.

The LEAP compilation flow is shown in Figure 4. The compiler gathers various LEAP memories instantiated in the user program and assembles them into a memory hierarchy, as in Figure 2. To efficiently utilize the bandwidth of multiple board-level memories on modern FPGAs, instead of building a single large central cache, LMC [11] treats each board-level memory resource as an independent cache managed by a separate controller hierarchy and assigns memory clients to the controller hierarchies in an application-specific fashion. Specifically, LMC measures the traffic sent from each memory client via program instrumentation and uses the measurement as feedback to construct a partitioned memory network, which balances the traffic across controllers. Figure 3 shows an example of the partitioned memory network constructed by LMC.

In both the baseline (Figure 2) and LMC-optimized memory hierarchies (Figure 3), various memory components are connected via LEAP rings [7]. LEAP originally opted for ring-based topologies

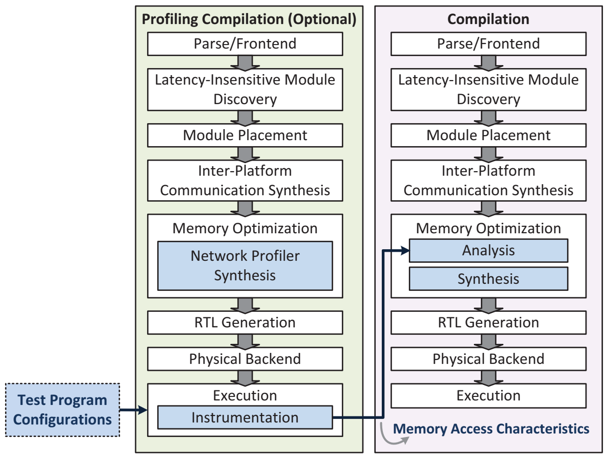

Profiling Compilation (Optional)
Parse/Frontend
Latency-Insensitive Module Discovery
Module Placement
Inter-Platform Communication Synthesis
Memory Optimization
Network Profiler Synthesis
RTL Generation
Physical Backend
Execution
Instrumentation
Test Program Configurations
Compilation
Parse/Frontend
Latency-Insensitive Module Discovery
Module Placement
Inter-Platform Communication Synthesis
Memory Optimization
Analysis
Synthesis
RTL Generation
Physical Backend
Execution
Memory Access Characteristics

Figure 4: The LEAP compilation flow [15][11], with our augmentations highlighted in blue. We extend the memory optimization phase introduced in LMC to construct more complicated cache networks.

because they are lightweight, largely symmetric, reasonably fair, and easy to assemble. However, as FPGAs and FPGA applications have scaled with Moore's law, the main flaw of ring-based memory networks has been increasingly exposed: latency.

In this work, we seek to alleviate the latency issue present in scaled out FPGA memory systems through the construction of program-optimized tree-based networks. Unlike ring-based networks, tree-based networks can be asymmetric: the compiler can choose both the radix of each interior node and the depth of a given memory client within the network. We extend LMC with algorithms that can synthesize tree-based networks tailored to the latency and bandwidth requirements of a program's memory clients, as measured by program instrumentation. Since the design space is large, we also introduce a network profiler to help explore our algorithmic choices.

3. RELATED WORK

Memory is fundamental to the performance of almost all computational systems. As such, memory systems have long been a focus of intense academic and industrial study. In general-purpose systems, memory architecture is usually determined through human implementation effort due to the high production volume of these systems and their symmetry. However, in lower volume architectures, like embedded SoCs, which have asymmetric use cases, design automation is often employed to optimize the memory system topology.

In the embedded domain, multiple accelerator devices are used to meet performance and energy targets. Multiple automated methods [16][17][18] for building memory networks incorporating such accelerators have been developed. These works propose the generation of custom memory topologies, generally consisting of a combination of shared buses and crossbars of various types based on communication patterns among the accelerators. As with our work, mixed-integer linear formulations have been used to optimize these topologies, given some performance characteristics and goals of the accelerators in the target system. Other works have considered the implementation of SoC-style memory networks in the context of FPGAs [19][20], in which optimization techniques are used to build custom crossbar and bus cascades. None of these works consider the construction of performance-optimized memory topologies for FPGA-based compute accelerators, in particular the case in which a single application may have many simultaneously active memory interfaces that must be balanced to achieve high performance.

A second major difference between our work and prior network synthesis studies lies in the choice of network topologies. Prior works focus almost exclusively on constructing SoC-style networks, which are intended to support memory accesses by a single accelerator and memory-mapped communications between accelerators ganged together to perform some task. This requirement results in very general communication topologies: shared buses and crossbars. We remark that, in FPGA-based compute accelerators, the tasks of communication and memory are usually separated. Communications are typically implemented directly and within the accelerator, while memory systems are confined to state storage. Leveraging this observation, we satisfy the memory needs of accelerators using simpler memory networks than contemplated in prior work, in turn improving key metrics such as area, frequency, and energy.

As FPGAs have grown in their capability as accelerators, several FPGA-specific memory system architectures have been proposed. In this paper, we adopt the LEAP memory and compiler as a base, but we believe most other architectures are sufficiently abstract to be compatible with our approach. CoRAM [6] advocates memory interaction using control threads programmed with a C-like language. CoRAM does not define the memory subsystem backing its programmer interface, and therefore could make use of our optimized memory networks. More traditional FPGA-based processor infrastructures [21][22][23], could also benefit from our work in low-latency memory networks as processors, and especially soft processors, are typically sensitive to memory access latencies. However, the processor memory behavior, as noted above, is typically symmetric when viewed across many workloads. Thus, this class of FPGA programs might not benefit from our optimizations.

Beyond these architectural efforts, researchers have also explored cache microarchitectures and multiple-level memory hierarchies on FPGAs [13][8][24][25]. These works generally assume a fixed, program-invariant memory topology, while our work focuses on optimizing the memory topology on a per-application basis.

4. MOTIVATING EXAMPLE

The advantage of customized memory networks is most salient for applications with a large number of asymmetric memory clients. One example is a high-performance hardware implementation of a *filtering algorithm* [26] for K-means clustering, a widely used machine learning technique for unsupervised partitioning of a data set. K-means clustering partitions a data set into K clusters such that each point belongs to the cluster with the nearest mean. The filtering algorithm prunes the search space of the nearest centers by organizing the data points in a binary search tree (a 'kd-tree' [26]) and finding nearest centers using a tree traversal.

In each iteration, the filtering algorithm traverses the tree starting from the root. Each tree node represents a subset of input data points and the algorithm propagates several candidates for the closest center to each subset down the tree. Our implementation uses three data structures: (i) A kd-tree that is built up from the data points and implemented as a pointer-linked binary tree. (ii) A stack that manages the tree traversal and is implemented as a pointer-linked list, whose head is modified by 'push' and 'pop' operations. (iii) Multiple sets of candidates for the closest center to a data subset. These candidate sets are of variable size and are created and disposed at runtime. The accesses to these data structures are essentially pointer chasing, which makes the execution time of the algorithm very sensitive to the memory access latency.

We parallelize the implementation by splitting the tree and the stack into $P = 8$ partitions and each partition maintains its own center sets. The computational kernels are implemented through high-level synthesis and connected to the LEAP memory system.

```
interface SERVICE_CLIENT_IFC#(type t_REQ, type t_RESP);
    method void sendRequest(t_REQ req);
    method t_RESP receiveResponse();
    method Bool requestNotFull();
    method Bool responseNotEmpty();
endinterface

interface SERVICE_SERVER_IFC#(type t_REQ, type t_RESP,
                              type t_ID);
    method void sendResponse(t_ID client, t_RESP resp);
    method t_REQ receiveRequest();
    method Bool responseNotFull();
    method Bool requestNotEmpty();
endinterface
```

Figure 5: The abstract interfaces of service connections

We instantiate a LEAP private memory for each partition and data structure type, resulting in 24 LEAP memories to store the three different types of data structures. In the baseline LEAP memory hierarchy, all 24 memory clients are connected on a single ring, introducing long network latency. Even if we apply client partitioning mechanisms introduced in LMC on an FPGA with multiple on-board memories, the network latency impact is still significant. To improve performance we need to build a cache network with better scalability.

In addition, we observe that memory clients in the filtering algorithm have different behavior and some are more sensitive to latency than others. For example, stack accesses have very high data locality and all hit in a small first-level cache. Since none of stack access requests reaches the memory network, performance for stack accesses is insensitive to network latency and topology. On the other hand, the LEAP memories storing tree nodes send many read requests to the memory network, because the tree node structures are large, have low data locality, and do not fit in first-level caches. As a result, these memory clients are sensitive to the network latency increase. Increasing network latency for these clients has a significant impact on program performance. To achieve high performance, a program-optimized cache network should provide shorter network latency to the memory clients storing tree nodes by placing these clients closer to the memory controller. In Section 8, we will show that our optimized cache network provides a 44% performance gain over the baseline LEAP memory hierarchy and a 18% performance gain over the partitioned network generated by LMC.

5. COMMUNICATION ABSTRACTION

To automate the construction of program-optimized cache networks, we first introduce a new communication abstraction enabling a clean separation between the functionality of the cache network and physical topologies. This abstraction, which we call a service connection, is designed for centralized services in which a controller takes requests from multiple clients and replies, if necessary.

The service connection abstraction provides clients and servers with request-response-based interfaces as shown in Figure 5. The abstract interfaces allow compilers to construct various network topologies underneath. Figure 6 shows an example of a connected service with three clients and a server, which are instantiated by specifying a service name ("MEM" for example). Semantically, each client is connected to a server with a matched service name via two in-order channels: one for requests, and the other for responses. At compile time, the compiler gathers clients and servers with the same service name, assigns each client with a unique ID, and then constructs an optimized physical network. Requests sent from each client are tagged with the client's ID. The server receives requests from clients, processes the requests, and then sends responses back to the requester by specifying the requester's ID.

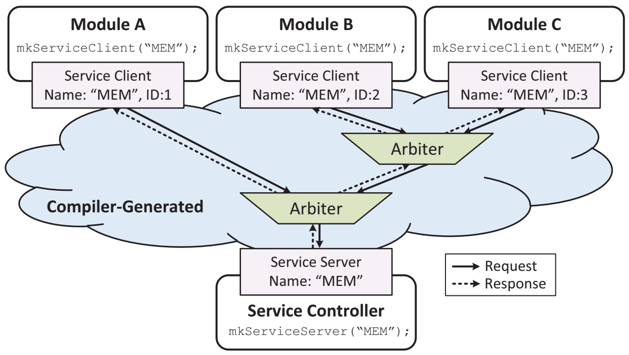
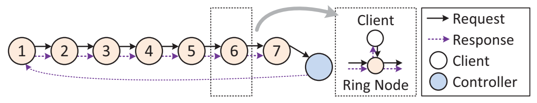
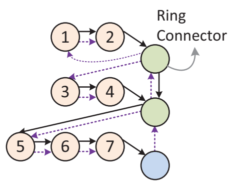


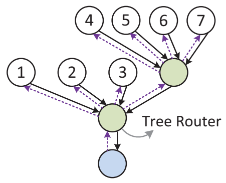
Figure 6: Communication abstraction for centralized services



(a) Single Ring



(b) Hierarchical Ring



(c) Tree

Figure 7: Examples of compiler-generated network topologies

This network construction strategy can also be applied to services with multiple servers. For example, the LEAP private memory service may have multiple private memory controllers to manage accesses to multiple on-board memories [11]. In this case, the compiler constructs a separate network for each server and each memory client can be connected to one or multiple servers.

6. NETWORK TOPOLOGIES

With the service connection abstraction, which merely defines the endpoint interfaces, the compiler is free to construct any network topology that connects service clients to their servers. To explore the design tradeoffs, we design the compiler to construct three different types of network topologies: a single-ring, a hierarchical-ring, and a tree network, as shown in Figure 7. Figure 7a is an example of a single-ring network, which works the same as the original LEAP rings. Physically, this network consists of two linear networks: one delivers requests from clients to the controller, and the other delivers responses from the controller to clients. Ring nodes check the requester ID of every incoming response packet and then decide whether to forward the packet to the local client or on the ring. Figure 7b shows an example of a hierarchical-ring network, which consists of multiple levels of rings and ring connectors. Similar to a ring node, a ring connector decides which ring to forward responses by checking the requester ID tagged with the response. The ID of each client is carefully assigned by the compiler in a sequential order, which makes response forwarding much easier at ring connectors. In both single-ring and hierarchical-ring networks, request and response packets are routed in a way so that clients on the same ring observe the same round-trip delay.

Figure 7c is a tree network. Each client is a leaf node and the controller connects to the tree root. The root node and the interior nodes of the tree are tree routers. A tree router forwards requests from its

Algorithm 1 Arbiter with Bandwidth Control

```
1: procedure REQUESTSCHEDULING(childList, bandwidthList)
2:     histList ← 0    ▷ Initialize each child's history bits to be zero
3:     while True do
4:         activeChildren ← ∅    ▷ Children with requests ready
5:         hungryChildren ← ∅    ▷ Children with unmet bandwidth targets
6:         priorityChildren ← ∅    ▷ Children without bandwidth limits
7:         for $i = 1, 2, \ldots,$ LENGTH(childList) do
8:             $c$ ← childList[$i$]
9:             if $c$ has requests ready to send then
10:                activeChildren ← activeChildren ∪ {$c$}
11:            ▷ hist: number of requests forwarded in the past period
12:            hist ← GETNUMOFONES(histList[$i$])
13:            if hist < bandwidthList[$i$].value then
14:                hungryChildren ← hungryChildren ∪ {$c$}
15:            if bandwidthList[$i$].limit ≠ True then
16:                priorityChildren ← priorityChildren ∪ {$c$}
17:        if activeChildren ∩ hungryChildren ≠ ∅ then
18:            candidates ← activeChildren ∩ hungryChildren
19:        else if activeChildren ∩ priorityChildren ≠ ∅ then
20:            candidates ← activeChildren ∩ priorityChildren
21:        ▷ Select the winner from candidates using round-robin
22:        winner ← ROUNDROBIN(candidates)
23:        Forward a request from winner to the output port
24:        for $i = 1, 2, \ldots,$ LENGTH(childList) do    ▷ Update history bits
25:            if childList[$i$] is winner then
26:                histList[$i$] ← histList[$i$] $\ll 1 + 1$
27:            else
28:                histList[$i$] ← histList[$i$] $\ll 1$
29: end procedure
```

children to its parent node using an arbiter, which is a K-to-1 MUX with bandwidth control, where K is the number of children. Algorithm 1 describes how the arbiter schedules requests from multiple children to the parent node given the bandwidth allocation information of each child. The bandwidth allocation information contains the bandwidth target, which is the number of requests that need to be served within a fixed period of time, and the bandwidth upper limit, which indicates whether the arbiter is allowed to forward requests from the child after its bandwidth target is met. The arbiter first forwards requests from hungry children whose bandwidth targets have not been met yet. If there are no hungry children, the arbiter then forwards requests from children which do not have bandwidth upper limits. If there are multiple candidates, a round-robin algorithm is used to select a winner. The tree router is also responsible for forwarding responses from the parent node to its children based on the requester ID tagged with the response.

The three kinds of network topologies shown in Figure 7 implement different cost-performance tradeoffs. The single-ring network has low design complexity but introduces long network latency when there is a large number of clients. Compared to the single-ring, the hierarchical-ring network has better network scalability with slightly more area overhead introduced by ring connectors. The tree network has the lowest network latency among the three networks but a tree router is much more complicated than a ring connector, especially when the tree router has a large number of children.

Constructing an optimized memory network usually involves the exploration of cost-performance tradeoffs, which may vary from application to application. For example, a program with high data locality may only require a simple cache network, since most memory requests are served in first-level caches, while a program with lower data locality and more memory clients may prefer a tree-based cache network, which has better scalability. In addition, even if a topology has been selected, placing memory clients in the network may still be challenging when the memory clients of the target program have different latency and bandwidth demands due to asymmetric memory access behavior. A deeply-pipelined client may be able to tolerate longer network latency but have larger bandwidth demands, while a latency-oriented client may be more sensitive to network latency but have smaller bandwidth demands. Therefore, it is essential to develop mechanisms to automate the design space exploration.

7. COMPILER OPTIMIZATION

To automate the construction of memory networks optimized for a particular application, we extend LMC with more detailed program introspection and optimization algorithms for selecting an optimal cache network. Figure 4 shows the extended LEAP compilation flow, which optionally includes profiling compilation to explore the network design tradeoffs on a per-application basis. During profiling compilation, a network profiler is constructed with program instrumentation hardware. The target program is then run with several test configurations to obtain the runtime information of memory access behavior for each memory client. Finally, the target program is recompiled and an optimized memory network is constructed based on the program instrumentation results. Section 7.1 describes the proposed network profiler and how we characterize the memory access behavior for each memory client. Section 7.2 introduces the algorithms by which the compiler synthesizes an optimized network.

7.1 Program Introspection

The goal of a profiling compilation is to understand the memory access behavior of each memory client and to examine the network design tradeoffs for the target application. To evaluate the performance impact of various network configurations, we could build the system several times, each with a different network implementation. However, this approach is very time consuming since each compilation requires full FPGA synthesis, placement, and routing. To facilitate the design space exploration, we design a dynamically-configurable, application-specific network profiler to emulate different network configurations in a single compilation. This network profiler needs not offer optimal performance. It is a measurement tool and can be used to characterize the latency and bandwidth requirements of each memory client in the target application.

Figure 8 shows an example of the application-specific network profiler, which is automatically constructed during profiling compilation. Program instrumentation logic is inserted at each memory client to monitor various runtime memory access properties, including the total number of requests sent from the client, the average request rate, and the average request queueing delay. The request and response ports of each memory client are connected with FIFOs that can be dynamically configured to delay requests/responses for a certain number of cycles. These *latency FIFOs* can be used to measure each client's network latency sensitivity as well as to emulate different network topologies. To configure the delay of the latency FIFOs at runtime, we utilize the LEAP dynamic parameter service, which allows parameter values to be overwritten at runtime through command-line switches.

In the network profiler, each on-board memory is managed by a separate controller hierarchy, and each memory client is connected to all controllers via memory interleaver logic. A memory interleaver partitions a single private memory's address space into multiple interleaved regions [11]. The size of each interleaved region is also dynamically configurable. Each memory client can be assigned to one of the controllers or to multiple controllers with variable-sized interleaved regions at runtime, enabling the performance evaluation of different partitioning algorithms.

The network profiler represents an ideal network with single-cycle latency by directly connecting each memory client to each

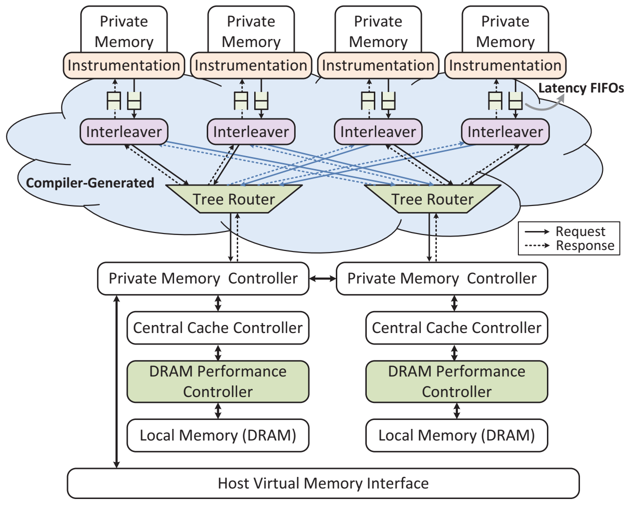


Figure 8: An application-specific network profiler with instrumentation logic and latency FIFOs inserted at each memory client to characterize the client's bandwidth and latency properties.

controller through N-to-1 tree routers, where N is the number of memory clients in the system. For each tree router in the profiler, the assigned bandwidth allocation information, which contains the bandwidth target and the bandwidth upper limit for each client as described in Section 6, is also dynamically configurable, enabling the evaluation of different bandwidth allocation strategies.

The latency FIFOs, the tree routers, and the memory interleavers in the network profiler are all dynamically configurable, enabling the profiler to emulate the performance of different network topologies, such as a singe ring, hierarchical rings, and tree-based networks. For example, to emulate a tree-based network, the delay of latency FIFOs for each client is configured based on the distance between the client and the controller in the target tree network.

This network profiler is a measurement tool that characterizes the network requirements for a particular application and therefore does not need to hit the application's target frequency. Instead, the profiler is usually constructed at a much lower frequency, making the construction of large single-cycle tree routers feasible. In theory, if the network delay and bandwidth allocation for each client are correctly modeled and if the frequency of the profiler is properly scaled down from the application's target frequency, the profiler can achieve very high accuracy. This means the runtime cycle count obtained from the profiling system can be very close to that from the final system with actual network implementation running at target frequency. However, we find that it is difficult to slow down the DRAM operating frequency in the profiling system, resulting in inaccurate performance emulation. To resolve this issue, for each DRAM bank we insert a DRAM performance controller that matches the DRAM latency and throughput to the profiled network.

7.2 Optimized Tree Networks

Armed with the knowledge of program behavior obtained from the network profiler, we can proceed to build a program-optimized cache network. Unless the target program is insensitive to network latency or requires very high operating frequency, the compiler prefers a low-latency tree-based network. The compiler constructs optimized cache networks through three stages: client partitioning, tree topology selection with client placement, and bandwidth allocation.

The compiler first determines the partitioning of the memory clients by passing the profiling results to the partitioning algorithm developed in LMC, which balances the total traffic across controller networks. After partitioning, the next step is to determine the best tree topology of each controller network.

The goal is to construct a tree network that minimizes the network latency impact on program performance. In a tree network, each client is viewed as a tree leaf node, and the controller is the tree root. Ideally, the best solution is to construct a depth-one tree, where the root directly connects to all leaf nodes. However, the complexity of the tree router may result in frequency degradation when there is a large number of leaves. Therefore, to maintain the target frequency, the number of children per tree node is constrained to be no greater than K. To construct an optimal tree network, we need to model the importance of each client, i.e., the impact of placing each leaf node on the overall program performance. This importance factor, which we refer to as latency sensitivity, may be affected by various memory access characteristics of the target client, such as the hit rate of the first-level cache, the memory request rate, or the depth of the computational pipelines. Instead of building a complicated performance model, we define a weight function w_{nd} to be the performance impact introduced by the nth leaf node if placed at tree depth d. This weight function is measured using the network profiler with the following expression:

$$w_{nd} = \frac{\text{runtime(tree with leaf } n \text{ at depth } d \text{ and rest at depth } 1)}{\text{runtime(depth-one tree)}}$$

With this weight function, the original performance maximization problem can be modeled as an optimization problem in which the total tree weight is minimized. Given a leaf node, its weight values are non-decreasing as the tree depth increases. This sets an upper bound for the maximum tree depth given K and the number of leaf nodes. To facilitate problem formulation without the loss of generality, we assume all non-leaf nodes must have exactly K children based on the following two observations: (i) A tree with maximum tree depth D is never optimal if any of the non-leaf nodes at depth $d < (D-1)$ has fewer than K children, because the total tree weight can be decreased by moving a leaf at larger depth to be the child of that node. (ii) We can add dummy leaf nodes with zero weight values so that the non-leaf nodes at depth $D-1$ also have K children, and the dummy leaves would be placed at depth D.

Suppose we are given the number of leaf nodes N, the maximum number of children per node K, the maximum tree depth D, and the weight w_{nd} of placing the nth leaf node at depth d for each $n = 1,\dots,N, d = 1,\dots,D$. We can formulate the topology synthesis problem as an integer linear programming (ILP) problem with the following decision variables for each $n = 1,\dots,N, d = 1,\dots,D$:

$$\begin{aligned} \lambda_{nd} \in \{0,1\} &\quad : \quad \text{whether the } n\text{th leaf is at depth } d \\ x_d \in \mathbb{Z}_{\geq 0} &\quad : \quad \text{number of leaf nodes at depth } d \\ y_d \in \mathbb{Z}_{\geq 0} &\quad : \quad \text{number of non-leaf nodes at depth } d \end{aligned}$$

where $\mathbb{Z}_{\geq 0}$ is the set of nonnegative integers. The problem can be stated formally as:

$$\begin{aligned} \text{minimize} \quad & \sum_{n=1}^{N}\sum_{d=1}^{D} w_{nd}\cdot\lambda_{nd} \\ \text{subject to:} \quad & \sum_{d=1}^{D}\lambda_{nd} = 1 \quad (n = 1,2,\dots,N) \\ & x_d = \sum_{n=1}^{N}\lambda_{nd} \quad (d = 1,2,\dots,D) \\ & y_0 = 1, \quad y_d + x_d = K\cdot y_{d-1} \quad (d = 1,\dots,D) \\ & \lambda_{nd} \in \{0,1\} \quad (n = 1,\dots,N; d = 1,\dots,D) \\ & x_d \in \mathbb{Z}_{\geq 0}, \quad y_d \in \mathbb{Z}_{\geq 0} \quad (d = 1,\dots,D) \end{aligned}$$

Algorithm 2 Construct a Minimum Weight Tree using DP

```
1: procedure TreeConstruction($N, D, K, \{a_n\}$)
2:     Sort $\{a_n\}$ so that $a_1 \le a_2 \le \cdots \le a_N$
3:     $V \leftarrow$ inf                                        ▷ $V[d][b][m]$: costs
4:     for $d = D, D-1, \ldots, 1$ do                            ▷ Base case: $d = D$
5:         for $b = K, 2K, \ldots, \min(\lceil \frac{N}{K} \rceil, (K-1)^d) \cdot K$ do
6:             for $m = 1, 2, \ldots, N$ do
7:                 if $b \ge m$ then                   ▷ Place all $m$ leaves at depth $d$
8:                     $V[d][b][m] \leftarrow d \cdot \sum_{n=1}^{m} a_n$
9:                 else if $d \ne D$ then
10:                    $V[d][b][m] \leftarrow$ FindMin($V, K, \{a_n\}, d, b, m$)
11: end procedure
12: function FindMin($V, K, \{a_n\}, d, b, m$)
13:     return $\min_{x=0,1,..,b\text{-}1} (d \cdot \sum_{n=m\text{-}x+1}^{m} a_n + V[d{+}1][(b\text{-}x)\cdot K][m\text{-}x])$
14: end function
```

For each leaf node n, if its weight values can be approximated as an affine function of depth d: $w_{nd} = a_n \cdot d + c_n$, $a_n \ge 0$, the tree topology synthesis problem can be solved using dynamic programming (DP), reducing the problem complexity to polynomial time. We first sort and re-index the leaf nodes so that $a_1 \le a_2 \le \cdots \le a_N$. Under this assumption, there exists an optimal tree in which the depth of node n is nondecreasing in n. Indeed, it is straightforward to verify that if there exists a pair of nodes whose depths are out of order, switching the positions of these nodes would result in a tree with a smaller total weight. This optimality condition allows us to decompose the problem into subproblems at each depth d. Let $V(d, b, m)$ denote the total weight of nodes at depth d or greater in the optimal tree that has m leaf nodes at depth d or greater, and b nodes (including leaf and non-leaf nodes) at depth d. The subproblems can be solved recursively as described in Algorithm 2, and the optimal number of leaf nodes at each depth can be found easily by backtracking the optimal solutions of each subproblem.

After the compiler constructs the optimal tree topology, which minimizes the network latency impact on performance, the final step is to determine the bandwidth allocation for each tree router. The compiler sets each leaf node's bandwidth target based on the client's request rate, which is measured by the program instrumentation logic, and the maximum request rate allowed by the central cache controller and the on-board DRAM. The bandwidth target for a non-root tree router is the sum of the targets of its children. When running multi-program applications, the bandwidth upper limits are set for leaves with large bandwidth demands, in order to control the fairness and prevent throughput-oriented applications from saturating the DRAM bandwidth and slowing latency-oriented applications.

8. EVALUATION

To evaluate the performance of our automatically-generated cache networks, we target a set of single-program and multi-program applications, on the Xilinx VC709 platform, which has two board-level 4GB DDR3 memories. We utilize Vivado HLS to transform HLS benchmarks into RTL implementations and employ Xilinx Vivado 2015.1 for all synthesis and physical implementation work. Also, we use the Gurobi optimizer [27] to solve the ILP problems. All resource utilization and clock rate results in this section are post-place-and-route results.

We examine the following single-program applications, which have a large number of asymmetric memory clients:

Filter: An HLS kernel that implements a filtering algorithm as described in Section 4. The implementation has 8 partitions ($P = 8$) to process independent subtrees and each partition uses 3 LEAP private memories to store different data structures.

Reflect-Tree: An HLS kernel that traverses a binary tree, heap-allocated data structure and swaps the left and right child pointers at each node, producing a mirrored tree in the memory. Similar to *filter*, the tree traversal is managed with a stack, which is implemented with a pointer-linked list. Each list node contains a pointer to a subtree. The head of the list is modified by push and pop operations, which ensures that the tree is traversed in a pre-order fashion. The program visits every node of the tree. Because of its pointer-chasing nature, the execution time of the benchmark is very sensitive to memory access latency. The implementation we target is split into 8 partitions and has 16 LEAP private memories in total.

We also set up the following multi-program applications to evaluate our bandwidth-controllable tree networks:

Cryptosorter-Filter: A multi-program application in which 4 *cryptosorters* [28] are constructed and scheduled to run with 8-partition *filter* ($P = 8$) at the same time. Each *cryptosorter* engine sorts an encrypted memory array, which is stored in a single LEAP private memory, using highly parallel merge-sort engines. It loads a large number of partially ordered lists then merges them using a high-radix sort tree. *Cryptosorter* is throughput-oriented and can consume almost as much bandwidth as the memory system provides. This multi-program application has 28 LEAP private memories: 4 from *cryptosorters* and 24 from *filter*.

Heat-Filter: A multi-program application that includes *heat* and *filter*. *Heat* is a two-dimensional stencil code modeling heat transfer across a surface. *Heat* can be split into multiple worker engines, each of which accesses a LEAP coherent memory. *Heat* is largely throughput-oriented. Workers traverse the shared two-dimensional space in fixed rectangular patterns. In this multi-program application, we construct *filter* with 4 partitions ($P = 4$), and the *heat* implementation has 8 worker engines. The shared memory space for *heat* is interleaved using techniques provided in [11] and is managed by dual coherence controllers. Each controller uses two private memories. This multi-program application has 16 LEAP private memories: 4 from *heat* and 12 from *filter*.

To show the performance benefit of program-optimized cache networks, we compare the performance of the compiler-generated networks with two previous solutions: (i) A baseline LEAP memory hierarchy with a single controller hierarchy to manage accesses to dual DRAMs. All memory clients are connected with a single ring. We refer to this implementation as *baseline*. (ii) A partitioned ring network constructed using LMC. This implementation has two controller hierarchies, and each controller connects to its clients via a single ring. We refer to it as the *single-ring* network configuration in this section. All of our implementations with compiler-generated networks have dual controller hierarchies and the memory clients are partitioned into two groups using the same partitioning mechanism as in the *single-ring* configuration.

8.1 Resource Utilization

Table 1 shows the resource utilization and maximum achievable frequency for different network primitives, including a ring node, which can also be used as a ring connector in hierarchical-ring networks, and K-to-1 bandwidth-controllable tree routers with varying K. As the number of input channels increases, the tree router complexity increases, resulting in larger frequency degradation. In order to maintain the operating frequency of the target application, the compiler needs to set an upper bound of K, limiting the maximum number of children for each interior node in the tree network.

To study the cost-performance tradeoffs of different network topologies, we extract the source code of the compiler-generated network from the target program and build the physical network alone with a standard FPGA tool flow. Table 2 shows the resource

Table 1: Resource utilization for various network primitives

Primitive		Slice LUTS	Slice Registers	f_{max} (MHz)
Ring Node/Connector		626	680	400
Tree Router	$K = 3$	437	577	179
	$K = 5$	826	811	143
	$K = 8$	1337	951	132
	$K = 16$	4501	2840	91
	$K = 32$	19390	11866	71

Table 2: Resource utilization for the cache network in $filter$

Configuration	Slice LUTS	Slice Registers	f_{max} (MHz)
Single Ring	17321	22968	400
Hierarchical Ring	18187	24811	400
Tree ($K = 3$, ILP)	9992	12914	179
Tree ($K = 6$, ILP)	7990	10760	139

utilization and frequency comparison of different network configurations for the cache network in *filter*. Given a fixed number of clients, compared to tree-based networks, ring-based networks can achieve much higher frequency but are less area efficient due to multiple message buffers in each ring node. If programs require high operating frequencies, hierarchical-ring networks, which trade area for maintaining high frequency and improving network scalability, are preferred; otherwise, the compiler would construct tree-based networks, which have lower latency and introduce less area overhead. Since the applications we study all run at a frequency below 130 MHz, as we will show in Section 8.2, tree-based networks with a smaller K are able to maintain the target frequency and provide better performance compared to ring-based networks.

8.2 Single-Program Applications

To construct a tree-based network that minimizes the network latency impact on program performance given an upper bound of K, the compiler first measures the latency sensitivity of each memory client using the network profiler. Figure 9 shows the latency sensitivity measurement of memory clients in *filter*. The data points are client weight values w_{nd} used in the ILP solution as described in Section 7.2. For each memory client in *filter*, its weight values form a straight line and therefore can be approximated as an affine function of tree depth, allowing the compiler to solve the tree construction problem using DP. The slope of each line represents the latency sensitivity, which is a_n in the DP solution described in Section 7.2. The memory clients that store stack data structures are not shown in Figure 9 because they have 100% hit rate in the first-level caches and therefore have zero latency sensitivity. The memory clients storing tree nodes have larger latency sensitivity due to a large number of nonparallel read misses, while the memory clients storing sets of center candidates have higher data locality and thus have latency sensitivity close to zero.

To show the effectiveness of our tree construction algorithms, we first compare the ILP and DP solutions with two other approaches in which a balanced tree with a minimum number of interior nodes is constructed. In a balanced tree, the sum of each leaf's depth is minimized, forming an optimal solution if clients have identical weights. *Balanced-tree (random)* first constructs a balanced tree and then randomly assigns clients to leaves. This approach is also used by our compiler when the profiling compilation is disabled and latency sensitivity of each client is unknown. *Balanced-tree (sorted)* is a greedy approach: it first determines a balanced tree topology and sorts the clients based on their latency sensitivity; then, it assigns latency-sensitive clients to leaves with a smaller tree depth.

Figure 10 shows the simulated performance of *filter* built with the four tree construction algorithms at a varying tree radix K. Since the

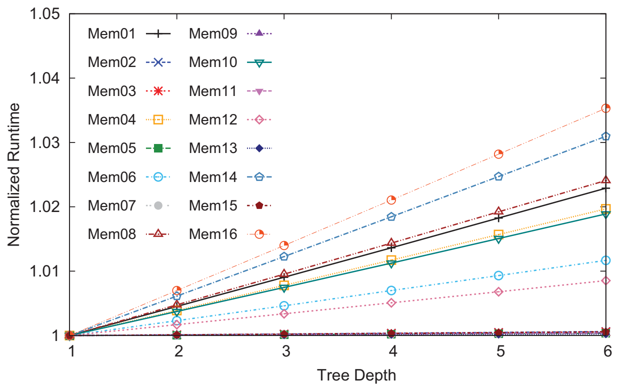


Figure 9: Latency sensitivity of memory clients in *filter*

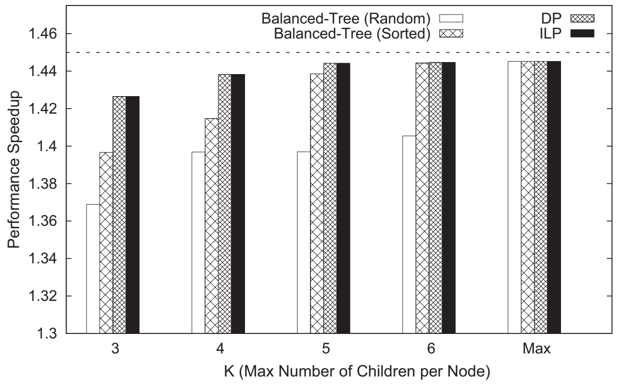


Figure 10: Simulated performance of *filter* with various tree construction algorithms. Performance speedup is calculated by comparing the runtime cycles measured from the network profiler to the runtime of the actual baseline implementation.

weight functions of *filter* clients are very close to affine functions, the tree constructed using DP is identical to that using ILP. As shown in Figure 10, these two solutions achieve good performance even when K is small. When K is larger than 4, the performance of the tree network constructed by DP and ILP is very close to the ideal network (where K = Max). The performance of *balanced-tree (sorted)* increases fast as K increases and reaches a nearly optimal value when $K = 6$, while the performance of *balanced-tree (random)* only slightly increases as K increases from 3 to 6.

We verify the simulated network performance by comparing it to the actual, physical implementation. Since the compiler-generated network module (with K less than 8 for tree-based networks) is not a frequency-limiting module for our test applications, we run all actual, physical implementations at the same frequency to make runtime cycles comparable. Figure 11 shows both the simulated performance and the actual performance of various network configurations for single-program applications. For each network topology, the network profiler is shown to have high accuracy: the performance difference is 1.1%, on average.

In addition, compared to other network topologies, our compiler-optimized tree networks achieve the best performance. For *filter*, the optimized tree network provides a 44% performance gain over the *baseline* and a 18% performance gain over the *single-ring* configuration. For *reflect-tree*, the tree network provides 47% speedup over the *baseline* and 16% speedup over the *single-ring* configuration. We also include a hierarchical-ring configuration, in which the compiler constructs a three-level hierarchical ring based on the latency sensitivity of each client. The hierarchical-ring solution achieves a

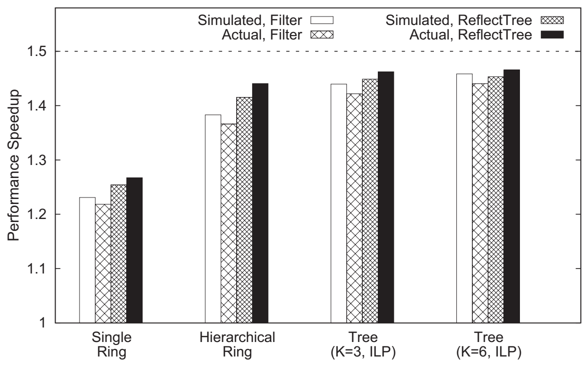


Figure 11: Performance comparison of various network configurations for single-program applications. Performance speedup is calculated by comparing the runtime cycles measured from the profiler and from the actual, physical implementations to the runtime of the actual baseline implementation.

12% performance gain over the *single-ring* configuration for *filter* and a 14% gain for *reflect-tree*, representing an effective approach to reduce the network latency impact at lower complexity.

8.3 Multi-Program Applications

We evaluate the performance of our compiler-optimized cache networks for multi-program applications by comparing the performance slowdown caused by resource sharing. We define the performance ratio r for each program as follows:

$$r = \frac{\text{Performance}_{MP}}{\text{Performance}_{SP}}$$

where Performance$_{MP}$ is the program performance when executing with other programs and Performance$_{SP}$ is the performance measured when executing alone. We also adopt the following fairness metric proposed in [29]:

$$\text{Fairness} = \frac{n}{\sum_{i=1}^{n} \frac{1}{r_i}}$$

where n is the number of programs and r_i is the performance ratio of the ith program. This fairness metric, which ranges from zero to one, is the harmonic mean of performance ratios.

Figure 12 shows the performance comparison of various network configurations when *filer* and *cryptosorter* are scheduled to run simultaneously. To make a fair comparison, we control the number of iterations *filter* executes so that *filter* and *cryptosorter* start and finish at the same time. The performance of *filter* is defined as the number of iterations executed in a fixed period of time. As shown in Figure 12a, *filter* performance slows down a little for *baseline* and *single-ring* configurations when *cryptosorter* is constructed on FPGA due to the increased network latency introduced by 4 additional clients. When *filter* is executing with *cryptosorter*, if without bandwidth control, the performance slowdown is over 50% because *cryptosorter* saturates the memory bandwidth, while our bandwidth-controllable tree network reduces the performance slowdown to 5% by limiting the bandwidth consumption of *cryptosorter* clients.

As shown in Figure 12b, when *filter* is constructed, the *cryptosorter* performance slowdown caused by 24 additional clients is larger for the *baseline* and *single-ring* configurations, while the performance of tree-based networks is much less sensitive to the increase of memory clients. When executing with *filter*, if without limitation on bandwidth consumption, *cryptosorter* performance does not slow down because *filter* only consumes little memory bandwidth. Adding bandwidth limitation to *cryptosorter* clients degrades the *cryptosorter* performance by 13% but achieves much better fairness as shown in Figure 12c.

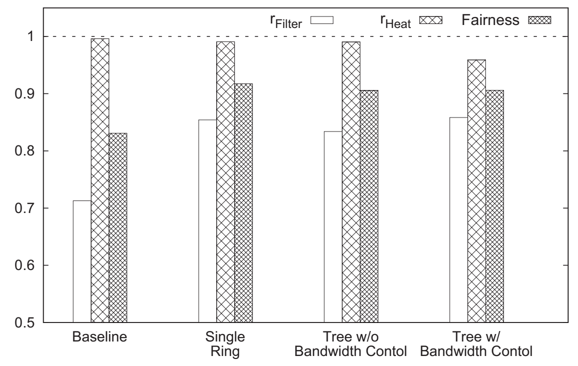


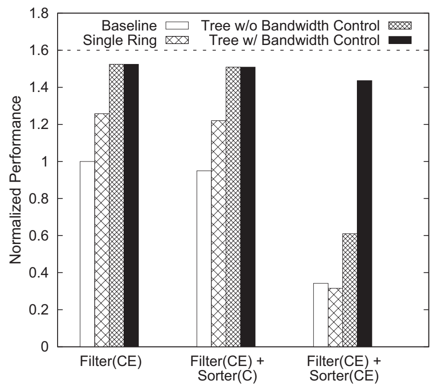
Figure 13: Performance comparison for the *heat-filter* application.

We also evaluate the performance ratios and fairness for *heat-filter* with different network configurations, as shown in Figure 13. Similar to *cryptosorter*, if without bandwidth limitation, *heat* does not slow down when executing with *filter*, which only has little bandwidth consumption. With partitioned networks, which provide larger memory bandwidth to the clients, the performance slowdown of *filter* is less than 15% when executing with *heat*, even without bandwidth control. The bandwidth consumption of *heat* is smaller than that of *cryptosorter* because of the higher data locality in coherent caches, resulting in a smaller memory bandwidth pressure. Therefore, all partitioned networks can achieve good fairness (above 0.9) when simultaneously executing *heat* and *filter*. Adding bandwidth limitation to *heat* clients in the tree network degrades *heat* performance by 4% and improves *filter* performance by 2%, achieving similar fairness as the tree network without bandwidth control.

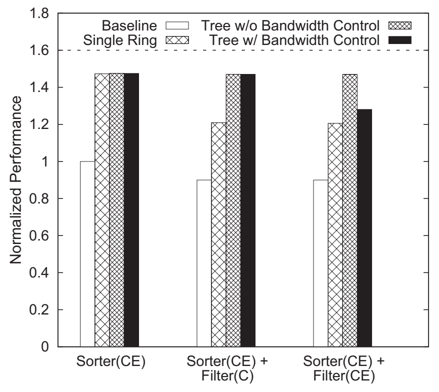
9. CONCLUSION

We have presented a feedback-driven compiler that automatically constructs memory networks optimized for the target application. In order to facilitate the design space exploration, we propose a dynamically-configurable network profiler that can be used to characterize the latency and bandwidth requirements of each memory client as well as to evaluate the performance impact introduced by different network topologies. Based on the profiling measurements, the compiler constructs an optimized cache network that minimizes the network latency impact on program performance. Experimental results show that our compiler-optimized network significantly improves the performance of applications that have a large number memory clients with asymmetric memory behavior: it provides a 45% performance gain over the baseline memory network and a 17% gain over the partitioned, ring-based network constructed by LMC, on average. In addition to single-program applications, we also examine a new set of workloads: multi-program applications, which we view as representative of future FPGA deployments. When multiple user programs are executing simultaneously, our compiler-generated network is shown to achieve good cross-application fairness through application-specific bandwidth control.

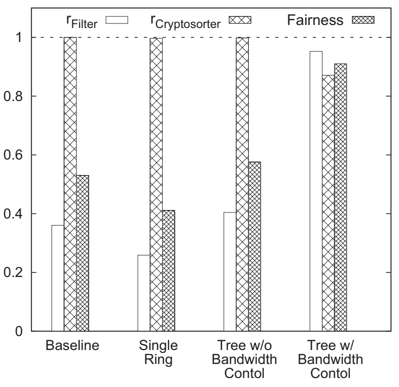
In this paper, we have demonstrated that applications with asymmetric memory clients can benefit from program-optimized memory networks. As modern FPGA platforms have begun to include asymmetric on-board memory controllers [30][12], one direction for future work is to explore resource-aware memory network optimizations for asymmetric memory controllers with different latency and bandwidth characteristics.



(a) *Filter* Performance



(b) *Cryptosorter* Performance



(c) Fairness

Figure 12: Performance comparison of various network configurations for the *cryptosorter-filter* application. (a) and (b) show the performance of *filter* and *cryptosorter* under different program configuration settings, where C indicates the program hardware is constructed on FPGA and E indicates the program is executed. For each program, the performance is normalized to the performance of the implementation where the program is constructed alone with the baseline memory network.

10. REFERENCES

[1] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *FCCM*, 2010.

[2] J. Cong, B. Liu, S. Neuendorffer, et al. High-level synthesis for FPGAs: From prototyping to deployment. *TCAD*, 30(4):473–491, 2011.

[3] A. Canis, J. Choi, M. Aldham, et al. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *TECS*, 13(2):24, 2013.

[4] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[5] R. Kirchgessner, G. Stitt, A. George, and H. Lam. VirtualRC: a virtual FPGA platform for applications and tools portability. In *FPGA*, 2012.

[6] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An in-fabric memory abstraction for FPGA-based computing. In *FPGA*, 2011.

[7] K. Fleming, H.-J. Yang, M. Adler, and J. Emer. The LEAP FPGA operating system. In *FPL*, 2014.

[8] J. Choi, K. Nam, A. Canis, et al. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *FCCM*, 2012.

[9] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer. Scavenger: Automating the construction of application-optimized memory hierarchies. In *FPL*, 2015.

[10] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides. Custom-sized caches in application-specific memory hierarchies. In *FPT*, 2015.

[11] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer. LMC: Automatic resource-aware program-optimized memory partitioning. In *FPGA*, 2016.

[12] Accelerating datacenter workloads. http://fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf.

[13] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP Scratchpads: Automatic memory and cache management for reconfigurable logic. In *FPGA*, 2011.

[14] H.-J. Yang, K. Fleming, M. Adler, and J. Emer. LEAP shared memories: Automating the construction of FPGA coherent memories. In *FCCM*, 2014.

[15] K. E. Fleming. *Scalable Reconfigurable Computation Leveraging Latency Insensitive Channels*. PhD thesis, MIT, Cambridge, MA, 2012.

[16] M. Jun, S. Yoo, and E.-Y. Chung. Mixed integer linear programming-based optimal topology synthesis of cascaded crossbar switches. In *ASP-DAC*, 2008.

[17] M. Jun, S. Yoo, and E.-Y. Chung. Topology synthesis of cascaded crossbar switches. *TCAD*, 28(6):926–930, 2009.

[18] A. Cilardo and E. Fusella. Design automation for application-specific on-chip interconnects: A survey. *Integration, the VLSI Journal*, 52:102–121, 2016.

[19] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo. Automated synthesis of FPGA-based heterogeneous interconnect topologies. In *FPL*, 2013.

[20] Y.-T. Chen and J. Cong. Interconnect synthesis of heterogeneous accelerators in a shared memory architecture. In *ISLPED*, 2015.

[21] E. Matthews, L. Shannon, and A. Fedorova. Polyblaze: From one to many bringing the Microblaze into the multicore era with Linux SMP support. In *FPL*, 2012.

[22] H. Lange, T. Wink, and A. Koch. MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In *DATE*, 2011.

[23] V. Mirian and P. Chow. FCache: A system for cache coherent processing on FPGAs. In *FPGA*, 2012.

[24] D. Göhringer, L. Meder, M. Hübner, and J. Becker. Adaptive multi-client network-on-chip memory. In *RECONFIG*, 2011.

[25] E. Matthews, N. C. Doyle, and L. Shannon. Design space exploration of L1 data caches for FPGA-based multiprocessor systems. In *FPGA*, 2015.

[26] T. Kanungo, D. M. Mount, N. S. Netanyahu, et al. An efficient k-means clustering algorithm: Analysis and implementation. *TPAMI*, 24(7):881–892, 2002.

[27] Gurobi optimization. http://www.gurobi.com.

[28] K. Fleming, M. King, M. C. Ng, A. Khan, and M. Vijayaraghavan. High-throughput pipelined mergesort. In *MEMOCODE*, 2008.

[29] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *CAL*, 10(1):4–7, 2011.

[30] Nallatech 510t FPGA accelerator. http://www.nallatech.com/nallatech-510t-fpga-datacenter-acceleration/.