LMC: Automatic Resource-Aware Program-Optimized Memory Partitioning

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

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

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

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

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

ABSTRACT

As FPGAs have grown in size and capacity, FPGA memory systems have become both richer and more diverse in order to support the increased computational capacity of FPGA fabrics. Using these resources, and using them *well*, has become commensurately more difficult, especially in the context of legacy designs ported from smaller, simpler FPGA systems. This growing complexity necessitates resource-aware compilers that can make good use of memory resources on behalf of the programmer. In this work, we introduce the LEAP Memory Compiler (LMC), which can synthesize application-optimized cache networks for systems with multiple memory resources, enabling user programs to automatically take advantage of the expanded memory capabilities of modern FPGA systems. In our experiments, the optimized cache network achieves up to 49% performance gains for throughput-oriented applications and 15% performance gains for latency-oriented applications, while increasing design area by less than 6% of the total chip area.

1. INTRODUCTION

FPGAs have become increasingly popular as accelerators because of their energy-efficiency and performance characteristics. To maximize efficiency and performance, FPGA programmers have traditionally utilized low-level primitives and programming models, explicitly customizing their implementation both to the target application and *to the target platform*. This approach, while effective, has made FPGA programs difficult to write, limiting both developer productivity and portability across FPGA platforms. As a result, recent research in both FPGA-oriented programming languages [26] [8] [5] [3] and architecture [20] [7] [13] has focused on raising the level of abstraction available to FPGA programmers, with the goal of reducing programmer design efforts.

High-level abstractions provide clearly-defined, generic interfaces that separate user programs from underlying infrastructure implementations. These fixed interface layers allow users to write portable programs, which can run on different FPGA platforms without redesigning the application code. Low-level platform implementation details are handled by a combination of compilers and system developers. Such abstractions present a significant opportunity for compilers since the extra resources available on modern FPGAs can be used to improve program performance without perturbing the original user program. This is especially the case for coarse-grained resources like board-level memories, the integration of which have traditionally required significant architectural consideration at design time.

To produce efficient, performant, portable FPGA designs, we argue that automated, resource-aware optimization of abstract interfaces is essential for modern FPGAs. In this work, we focus on one aspect of resource-aware optimization: FPGA memory systems. As the availability of more transistors makes it feasible to build both larger, bandwidth-hungry designs and the memory controllers necessary to feed them, modern FPGA boards have begun to include multiple DDR and HBM memories [2]. Moreover, the number of memory controllers appears to be increasing rapidly as vendors move to harden memory interfaces [1] [2]. To improve FPGA system performance, it is critical to enable integration of these increasingly rich and varied memory systems into user programs without drastically increasing design burden.

To allow programs to automatically make use of available memory resources, we propose the LEAP Memory Compiler (LMC), an augmentation to the LEAP compilation flow [13]. Unlike general-purpose processors where the memory hierarchy is fixed at design time based on a set of expected workloads, LMC tailors FPGA memory systems to different applications at compilation time based on the properties of those specific applications. LMC presupposes user programs described in terms of high-level memory interfaces [4] [29], which hide memory implementation details from application designers. In order to improve the performance of user programs, LMC incorporates several optimizations that take advantage of both interface abstraction and the availability of extra memory resources.

LMC operates in three phases: instrumentation, analysis, and synthesis. In the first phase, LMC injects instrumentation infrastructure into the baseline memory system. This instrumentation is used to collect runtime information about the way the program uses memory. Subsequently, LMC analyzes these metrics and applies various optimization techniques to improve the performance of the memory subsystem specific to the application and the target platform. Finally, LMC emits an optimized memory system implementation which passes through a standard tool flow to produce an FPGA image.

The main contribution of our work is in the automated optimization of memory systems. LMC produces as output an optimized

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPGA'16, February 21-23, 2016, Monterey, CA, USA

© 2016 ACM. ISBN 978-1-4503-3856-1/16/02...$15.00

DOI: http://dx.doi.org/10.1145/2847263.2847283

cache network, connecting single or multiple user-level caches to the available on-board memory resources. We introduce optimizations for both private and shared memories based on the available on-board memory resources in the system, the number of FPGAs, as well as the user program's bandwidth, which we derive from runtime instrumentation. In support of and in conjunction with our optimizing compiler, we introduce several new memory microarchitectures that enable existing client memory primitives to target multiple board-level memories.

For private memories, we partition the user-level private caches into disjoint groups and connect each group to a shared cache implemented on top of each on-board memory bank. Each partition is connected using a separate, dedicated network, which serves to increase memory throughput while simultaneously reducing memory latency. To improve memory bandwidth, LMC implements an intelligent memory interleaving mechanism, which enables individual private memories to utilize multiple board-level memories. In interleaved memory systems, portions of the memory address space are routed to different memory resources at a relatively fine grain.

For applications using shared memory, we apply memory interleaving techniques to partition the shared memory address space into multiple disjoint regions. We adopt a hierarchical-ring topology in the partitioned coherent cache network. This hierarchical topology reduces the latency of coherence messages, which is essential if designs are spread across several networked FPGAs. Since the coherent memory hierarchy is integrated with the private memory hierarchy, our private memory optimizations can be directly composed with our coherent memory optimizations to further improve overall system performance.

We evaluate LMC using both hand-assembled and HLS-compiled applications by targeting several single-board and networked multi-board FPGA deployments. Our optimizations cost less than 6% of the total chip area. For hand-assembled, throughput-oriented applications, the partitioned network achieves up to 49% performance improvement because of the increased memory bandwidth. For the HLS-compiled application, which involves pointer-chasing and is latency-oriented, the partitioned network provides about 15% performance improvement because of the reduced network latency.

2. BACKGROUND

LMC produces program-optimized memory hierarchies. To produce these hierarchies, LMC requires a collection of primitive elements from which a memory system can be constructed. As a base for LMC, we adopt the primitives and compiler of the LEAP [13] memory hierarchy. In particular, we leverage two prior memory primitives: LEAP private memories [4] and coherent memories [29]. Both of these memory primitives are built on top of LEAP's latency-insensitive channels [11], named communications primitives that are instantiated within user programs and implemented by the compiler at compilation time.

LEAP latency-insensitive channels provide named point-to-point communications for hardware programs. At compilation time, send and receive endpoints in the user program are matched, and a flow-controlled channel implementation is instantiated between them. Additionally, LEAP supplies a ring primitive, in which the compiler assembles all similarly named ringstops into a network. Unlike traditional hardware programming, LEAP named channels enable communication between any two points in a design without explicitly connecting wires through the module hierarchy. This channel construction primitive greatly simplifies tasks such as restructuring memory topologies and instantiating counters for feedback directed compilation. LMC makes use of channel and ring renaming as a mechanism for constructing application-specific memories.

```
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: A general memory interface for hardware designs.

LEAP private memories provide a general, in-fabric memory abstraction for FPGA programs. Programmers instantiate private memories with a simple read-request, read-response, write interface, shown in Figure 1. Each memory represents a logically private address space, and a program may instantiate as many memories as needed. Memories may store arbitrary data types and support arbitrary address space sizes, even if the target FPGA does not have sufficient physical memory to cover the entire requested memory space. To provide the illusion of large address spaces, LEAP backs the FPGA memory with host virtual memory, while FPGA physical memories, including on-chip and on-board memories, are used as caches to maintain high performance.

LEAP coherent memories [29] extend the baseline LEAP memory interface to support shared memory. Similar to LEAP private memories, LEAP coherent memories provide a simple memory interface and the illusion of unlimited virtual storage. In addition, LEAP coherent memories permit applications to declare multiple, independent coherent address spaces using a compile-time specified name. Cache coherency and weak consistency are maintained among coherent memories that share the same address space.

LEAP's memory system resembles that of general-purpose machines, both in terms of its abstract interface and its hierarchical construction. Like the load-store interface of general-purpose machines, LEAP's abstract memory interfaces do not specify or imply any details of the underlying memory system implementation, such as how many operations can be in flight and the topology of the memory. This ambiguity provides significant freedom of implementation to the compiler. For example, a small memory could be implemented as a local SRAM, while a larger memory could be backed by a cache hierarchy and host virtual memory. LEAP exploits abstraction to build complex, optimized memory architectures on behalf of the user, bridging the simple user interface and complex physical hardware. In this work, we leverage the freedom of abstraction to target systems with multiple on-board memory banks.

LEAP organizes the various memory primitives in the user program into a memory hierarchy with multiple levels of cache. Like memory hierarchies in general-purpose computers, the LEAP memory organization provides the appearance of fast memory to programs with good locality. 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 LEAP private memories both literally and figuratively: each LEAP shared address space is backed by two LEAP private memories as data and coherence ownership stores. As a result of this layering, both types of memories share large portions of the memory hierarchy. LEAP memory clients optionally receive a local cache, which is implemented using on-chip SRAMs. A snoopy-based coherence protocol is implemented to maintain cache coherency among LEAP coherent memory clients' local caches.

The board-level memory, which is typically an off-chip SRAM or DRAM, is used as a shared cache or central cache. The central cache controller manages access to a multi-word, set-associative board-level cache with a configurable replacement policy. Within the cache, each private memory space and shared memory domain is uniquely tagged, enforcing a physical separation. Previously, LEAP

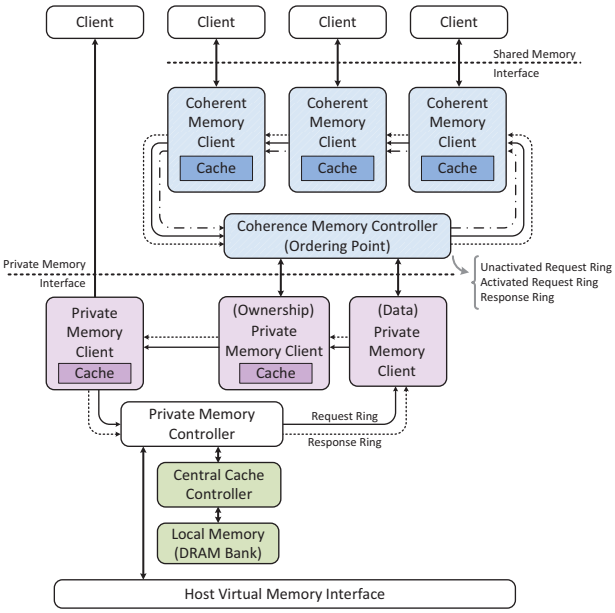


Figure 2: An example of LEAP memory hierarchy

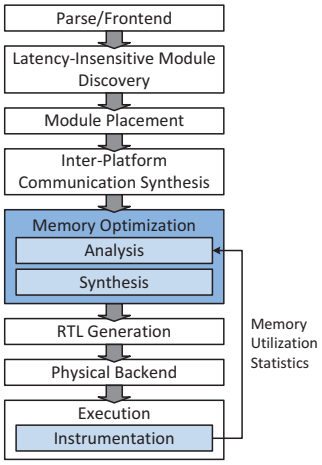


Figure 3: The LEAP compilation flow [14], with our augmentations highlighted. We add a memory optimization phase that performs memory network analysis and synthesis. Memory network analysis is optionally feedback driven.

aggregated all board-level memory resources into a single larger-capacity cache handled by a single central cache controller. In this work, we introduce the capability to treat each board-level memory resource as an independent cache, thereby creating the opportunity for program-specific memory hierarchy organization and partitioning, which we examine in detail in Section 5.

LEAP's compilation flow is shown in Figure 3. The compiler gathers the various LEAP memories in the user program and assembles them into a memory hierarchy, as in Figure 2. By default, the compiler assembles the various memory interfaces into ring networks. For example, coherent memories in each coherence domain are connected via three rings: the unactivated request ring, the activated request ring, and the response ring. These rings are used to implement the channels required by the snoopy coherence protocol. Lower levels of the memory hierarchy are self-assembled in a similar manner, integrating successive backing stores into the hierarchy. In this work, we augment LEAP's name-based network assembly mechanism with a compiler-generated indirection layer which influences the construction of the memory network based on memory system properties and program characteristics.

3. RELATED WORK

Our work presumes the existence of a basic memory abstraction for FPGAs, which we then back with a custom memory network. In this work, we build on the LEAP memory framework, but other frameworks are compatible with our approach. CoRAM [7] advocates memory interaction using control threads programmed with a C-like language. CoRAM does not define the memory hierarchy or network backing its interface, and therefore could use our optimization approach, a work that is ongoing. FPGA-based processor infrastructures [23] [21] [24], especially multicore systems, could also make use of our networks, though the benefit of optimizing such symmetric systems is less obvious.

Much recent work has gone into the microarchitecture of memory systems on FPGAs [9] [6] and the construction of multiple-level memory hierarchies [4] [6] [16] [22], all of which build user-level memory interfaces backed by some off-chip storage. Generally, the memory networks used by these systems are symmetrical and do not explicitly take into account program behavior.

Our technique of memory address interleaving across multiple memory controllers has been used extensively in general-purpose computer architecture to maximize available memory bandwidth and to simplify the microarchitecture of elements of the memory system, like coherency. Memory interleaving dates to early IBM mainframes. An example of memory interleaving is the SGI Challenge [15] line of supercomputers. Challenge's memory subsystem is comprised of multiple *leaf* memory controllers which were aggregated and interleaved in large systems using direct address partitioning. As transistor density has improved, most modern systems have opted to include multiple memory channels and controllers, most of which use various interleaving techniques to improve workload memory bandwidth and latency.

Hierarchical, ring-based coherence protocols also have a long history in computer architecture. Our work resembles some of the coherence architectures developed as part of the Hector project [10]. In this work, sets of processors are connected on coherent buses forming a collection called a *station*. The station controller serves to interface the processor collection to other stations on a local ring and filters messages not needed by the station. Local rings are aggregated to form a global ring. This arrangement of hierarchical rings helps reduce latency and overall network traffic by eliminating irrelevant messages on the local and station networks.

Since the memory networks of traditional computer architectures must be fixed or largely fixed at manufacture, previous work in the area of memory networks has largely focused on symmetric architectures, which are likely to handle a broad class of workloads reasonably well. Our per-program analysis permits us to leverage memory partitioning asymmetrically, if such asymmetry benefits a particular program.

4. LEAP MEMORY COMPILER

To automate the construction of optimized memory systems tailored for different applications, we extend the LEAP compilation flow [14] by adding a series of compilation phases, which we refer to as the LEAP Memory Compiler (LMC). Figure 3 shows the extended compilation flow incorporating LMC. LMC operates in three

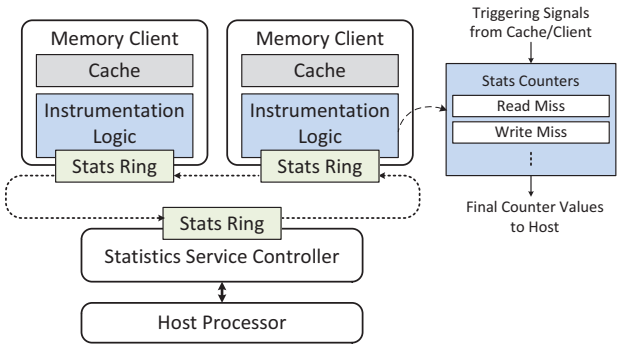


Figure 4: Program instrumentation built with the LEAP statistics collection service

phases: instrumentation, analysis, and synthesis. The combination results in an application-specific memory hierarchy.

The first phase of LMC is program instrumentation. For many FPGA applications, there are multiple memory clients in the system and the memory clients may have highly asymmetric behaviors and implementation needs. For example, a memory client producing a large number of cache misses within a short period of time may require more memory bandwidth from the next-level memory, while a memory client with high cache hit rate may be able to tolerate a larger miss penalty. Evenly distributing memory resources among asymmetric clients without knowing their memory utilization properties may cause bandwidth waste. To understand program behavior, static program analysis may be tractable, especially for HLS-compiled applications [28]. However, in order to target more general and more complicated programs including hand-assembled applications, we resort to FPGA-based runtime instrumentation.

Figure 4 describes our program instrumentation mechanism, which is built on top of the LEAP statistics collection service. Program instrumentation logic is inserted at each memory client to monitor various runtime memory utilization properties, such as the number of cache misses, the number of outstanding requests, and the request queueing delay. These instrumentation results are recorded in local counters at each memory client during program execution. We utilize the LEAP statistics service to collect instrumentation results at the end of the execution. LEAP statistics counters communicate using the standard LEAP named channels and rings. The LEAP compiler automatically connects the instrumentation logic to a centralized statistics controller via a LEAP latency-insensitive ring. When the controller receives a statistics-collection command from the host processor, it forwards the command to the clients and asks them to send back the instrumentation results. The host processor then records the collected results in a statistics file, which can be used in LMC's analysis phase in subsequent compilations.

During the analysis phase LMC analyzes program information, such as the number of memory clients, as well as platform information, including the number of FPGAs and the number of board-level memories. LMC then optimizes the memory hierarchy by assigning memory clients to available memory controllers associated with board-level memories. This phase is optionally feedback-driven: the instrumentation results obtained from previous program execution can be utilized for further optimizations, such as bandwidth-aware partitioning, which we will discuss in Section 5. The output of the analysis phase is an abstract representation of the memory hierarchy, which is passed to the synthesis phase.

The final phase of LMC is the synthesis phase, which produces an implementation of the application-specific memory hierarchy. To construct the synthesized memory hierarchy we leverage the

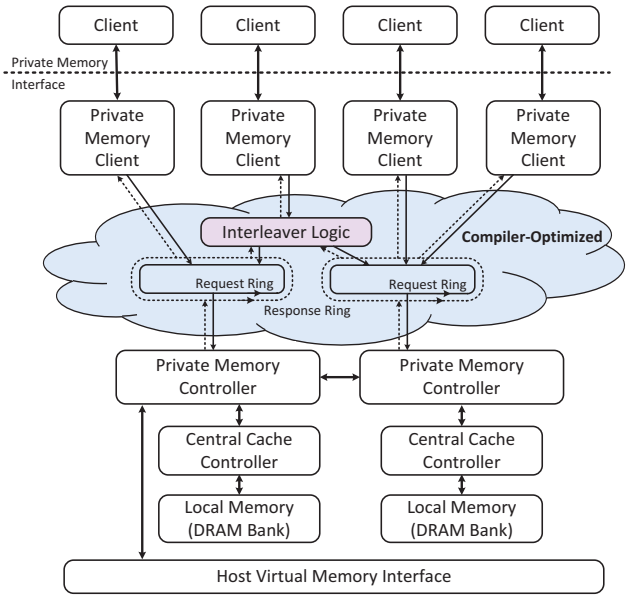


Figure 5: LEAP private memory system with a compiler-optimized cache network

name-based channel assembly mechanism provided by the baseline LEAP compiler, as mentioned in Section 2. Based on the abstract representation of the memory hierarchy obtained from the analysis phase, LMC generates a renaming function as the final output, which maps each client's memory connection to its automatically chosen memory controller. If no optimizations are applied in the analysis phase, we supply identity as the renaming function. After LMC is complete, we leverage the existing network synthesis capabilities of the RTL-generation phase in the baseline LEAP compiler to generate physical networks by matching the newly assigned channel names.

5. CACHE NETWORK OPTIMIZATION

This section introduces the analysis and mechanisms by which we synthesize platform-optimized and program-optimized cache networks for both private and shared memory services. In this work, we target the cases when there is more than one board-level memory controller. Operating from a high-level specification of a memory system, our optimizations produce cache networks that utilize increased memory bandwidth as well as reduce cache network latency.

5.1 Private Cache Network Optimization

To utilize the bandwidth of multiple on-board memories efficiently, we begin by constructing a central cache controller for each memory bank, as shown in Figure 5. The result is a set of distributed caches. Each central cache uses an on-board memory bank to store cache data and tags. The private memory controller, which offers a read/write interface to address spaces, is also duplicated per central cache. The memory controllers are responsible for accessing central cache banks for the associated private memory clients as well as communicating with the host memory backing store.

Since LEAP private memories have disjoint address spaces, they can be freely separated and mapped to different controllers and central caches without any changes in the private memory design. There are many possible mechanisms for assigning clients to memory controllers. A simple solution is random partitioning: we randomly separate private memory clients into (roughly) equal-sized groups, assign a memory controller to each group, and synthesize separate rings to connect all the nodes within the same group. For

Algorithm 1 Private Cache Network Partitioning

```
1:  procedure PARTITION(clientList, controllerList, statsFile)
2:      Initialize controller bandwidth to be zero
3:      if statsFile exists then                    ▷ Load-balanced partitioning
4:          clientBandwidth ← Parse statsFile
5:          while clientList not empty do
6:              s ← Memory client with maximum clientBandwidth
7:              c ← Controller with minimum bandwidth
8:              Connect s to c
9:              Add s's bandwith to c's bandwidth
10:             Remove s from clientList
11:     else                                        ▷ Random partitioning
12:         n ← Length of controllerList
13:         sLists ← Randomly separate clientList into n lists
14:         for i in 1 to n do
15:             Connect sLists[i] to controllerList[i]
```

applications with largely homogeneous memory clients, random partitioning effectively reduces network latency and balances the traffic among multiple controller networks. However, if the behavior of private memories is heterogeneous, i.e., the memory clients have different cache properties and different bandwidth demands, random partitioning may not achieve the best performance.

To improve performance of applications with heterogeneous clients we adopt feedback-driven load-balanced partitioning to separate memory clients into groups. Load-balancing is especially important for throughput-oriented applications, whose memory clients issue multiple outstanding requests to hide long-latency misses and therefore are more sensitive to the available bandwidth in the associated controller network. We utilize LMC's instrumentation mechanism to track the total number of messages sent from each memory client and use this metric as the first-order approximation of the client's bandwidth requirement. We then partition the memory clients based on their bandwidth estimation.

Algorithm 1 describes how we partition the private cache network on a single FPGA. We adapt the classical longest-processing-time (LPT) algorithm [17] to memory bandwidth partitioning. This algorithm approximates optimal load balancing by assigning new memory traffic to the least-loaded memory controller. To partition the cache network across multiple FPGAs, private memories are routed to one of the controllers on the same FPGA using our load balancing algorithm, avoiding long inter-FPGA communication latency.

A weakness of the load-balanced partitioning approach is that a single bandwidth-intensive memory client cannot utilize the full bandwidth of the memory system. To remedy this, LMC implements a memory interleaving mechanism that enables a single memory client to connect to multiple memory controllers. Memory interleaver logic is instantiated to partition a single private memory's address space into multiple, variable-sized interleaved regions. Requests targeting different regions are forwarded to different controller networks, allowing more physical bandwidth and more independent, parallel requests.

When memory clients consume a large amount of bandwidth or when it is difficult to perform load-balanced partitioning, for example, when there is only one private memory in the system, LMC constructs interleaved memories. Individual clients accessing interleaved memories are mapped to multiple controllers by injecting memory interleaver logic. Private memory interleaving is combined with the partitioning method described in Algorithm 1: LMC first deals with the memory clients whose memory needs to be interleaved, connecting them with multiple controllers, and then apportions the remaining non-interleaved memory clients.

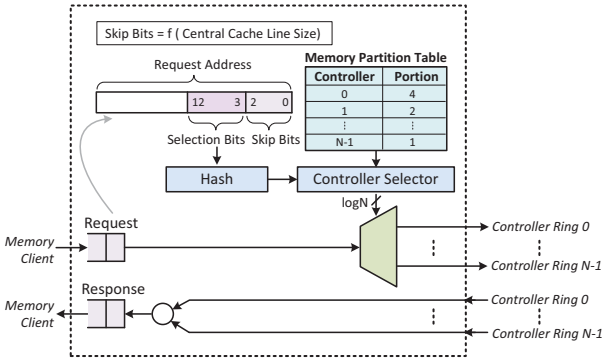


Figure 6: LEAP private memory interleaver logic microarchitecture

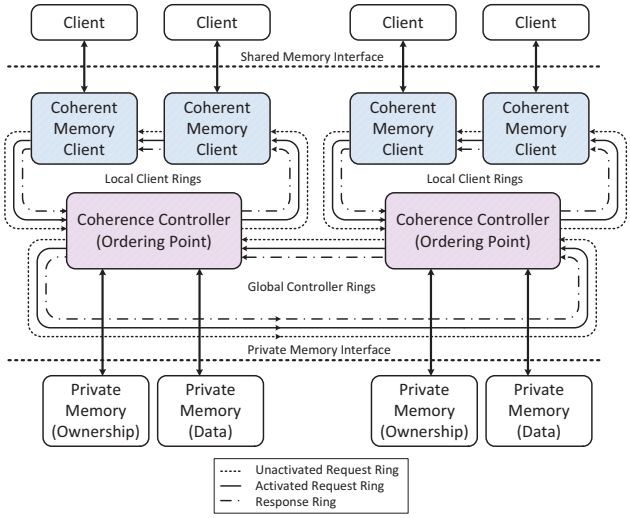


Figure 7: LEAP coherent memories with dual coherence controllers

Figure 5 shows an example of a compiler-optimized cache network with two on-board DRAM banks in the system. In this example, there is one private memory that connects to two controller networks via the memory interleaver logic. Figure 6 shows the microarchitecture of the private memory interleaver logic. When the memory interleaver receives a request from the associated private memory, it forwards the request to one of the controller rings based on the target request's word-level address. We route consecutive addresses to the same controller to take advantage of spatial locality available at the central cache: each central cache line is comprised of multiple private memory's cache words. We use the mid-order bits in the address field and the memory partition table to select the destination controller. The memory partition table records the portion of the address space assigned to each controller. The address space can be split into non-equal-sized banks, enabling fine-grained load-balanced partitioning. Since some strided access patterns may introduce controller selection conflicts and cause serialization at the memory controllers, we introduce a hashing function to apportion requests in a static but random fashion in order to balance requests between controllers.

5.2 Coherent Cache Network Optimization

Since LEAP coherent memories are built on top of LEAP private memories, applications that use LEAP coherent memories can benefit from previously described private cache partitioning optimizations. However, the optimization space is limited, since each

coherence domain only uses two private memories as data and ownership stores (see Figure 2).

To efficiently utilize the underlying memory resources in the case of coherent memories, we introduce a memory interleaving technique for a coherence domain. Coherent memory interleaving is conceptually similar to the private memory interleaving technique described above. The coherent memory address space is partitioned into disjoint regions and assigned to a distributed set of coherence controllers, each handling coherence for a separate region. Figure 7 shows an example of the partitioned coherent cache network that connects four coherent memory clients in a single coherence domain to dual interleaved coherence controllers. Coherent caches and interleaved controllers are connected via a hierarchy of ring networks, which reduces network latency and improves scalability. Although similar to the private memory optimization, the separation of the address spaces for coherent memories is complicated by the need to maintain coherency guarantees.

Within an interleaved coherence domain, each coherence controller is responsible for a portion of the domain address space. The controller connects to the next-level memory by instantiating two LEAP private memories, one to store data and the other to track coherence ownership information for the associated address region. The controller snoops every local request, whose target block address belongs to the controller's associated memory region, and responds to the requester if none of the coherent cache clients owns the data block.

Each controller has an address mapping function that determines whether an incoming request is local or not. The address mapping function is also responsible for converting the address of an incoming request from the global address space to the controller's local address space before the controller accesses the next-level memory. For coarse-grained memory interleaving, the address mapping function can be as simple as an address range filter. For fine-grained memory interleaving, the mapping function can be implemented as a look-up table.

Each coherence controller also serves as a distributed ordering point. In the original ring-based snoopy protocol [29], there is a single, global ordering point on the memory network (see Figure 2). To ensure all coherent memories see coherence requests in the same order, all requests must first go to the global ordering point before being broadcast in a global order, creating congestion at the single ordering point. To improve network performance for an interleaved shared memory system, each coherence controller gathers requests that are local to its memory bank and broadcasts them on the ring separately. Requests targeting different memory banks may be seen in a different order by different coherent memory clients. Coherency is maintained under this optimization because all clients agree with the ordering of operations on a single memory location. As for memory consistency, which specifies the memory ordering behavior for operations on multiple memory locations, coherent memory clients in the original protocol [29] perform out-of-order execution to achieve higher parallelism and thus only provide weak consistency guarantees. Introducing multiple ordering points does not weaken memory consistency. The LEAP coherent memory interface supports fences which can be invoked by clients that require stronger memory consistency to ensure the ordering of operations.

To improve network bandwidth, we partition the original ring network into hierarchical rings as shown in Figure 7. As a baseline, coherent memory clients are partitioned into equal-sized groups in lexical order, but other feedback driven partitioning algorithms can also be applied. Coherent memory clients in the same partition are connected to one of the coherence controllers via local client rings, and the controllers are connected together with global controller rings. As in the original protocol, three LEAP latency-insensitive rings are constructed for three types of coherence messages in each local and global ring network to prevent deadlocks: the unactivated request ring, the activated request ring, and the response ring. The global request and response rings can be viewed as express links that shorten the longest distance between the responder and requester. When memory clients are spread across multiple FPGAs, the hierarchical ring structure further improves network latency since the frequency of long-latency inter-FPGA communication is reduced.

6. EVALUATION

The majority of our evaluation targets the Xilinx VC709 platform. The Virtex-7 FPGA on the VC709 includes two physical memory controllers, each connected to 4GB DDR3 memories. We use these to implement two board-level caches per VC709. We also test two networked FPGA deployments: a dual VC707 and a dual VC709 configuration, to demonstrate how our techniques can be applied to cloud-based networks of FPGAs. We network our FPGAs using two bidirectional 10Gbps SERDES channels. Frequencies are normalized to 100MHz on all platforms to ensure performance results are comparable. For the HLS benchmark [27], we utilize Vivado HLS. We make use of Xilinx Vivado 2015.1 for all synthesis and physical implementation.

We examine a set of benchmarks with different memory access patterns in order to evaluate the benefit of LMC:

Memperf: A kernel that measures the performance of the LEAP memory hierarchy by testing various data strides and working set sizes on a single private memory. *Memperf* is throughput-oriented and can issue as many outstanding requests as the memory system permits.

Heat: A two-dimensional stencil code that models heat transfer across a surface. *Heat* is embarrassingly parallel and can be divided among as many worker engines as can fit on the FPGA. From an algorithmic perspective, *heat* is also very regular: workers march over the shared two-dimensional space in fixed rectangular patterns. As such, *heat* is largely throughput-oriented with a strided access pattern, but with a high degree of locality. Each *heat* worker accesses a LEAP coherent memory, and these coherent memories are largely symmetric in their runtime behavior. However, *heat's* coherence controllers include private memory clients for data and ownership, and these are asymmetric: the data client uses ten times the bandwidth of the ownership client.

Heat operates on a parametric data size. In this work, we examine two data size parameterizations: 8-bit, which gives a degree of spatial locality in the coherent cache, and 64-bit, which has less locality.

Cryptosorter: *Cryptosorter* [12] sorts a set of encrypted memory arrays using highly parallel merge sort engines. *Cryptosorter* loads a large number of partially ordered lists in a streaming fashion and then merges these lists within the fabric using a high-radix sort tree. *Cryptosorter* is throughput-oriented and can be scaled to consume almost any amount of bandwidth. Since the lists to be sorted are random, the access pattern of *cryptosorter* is also somewhat random. The merge operation and, therefore, the memory access behavior of *cryptosorter* is related to sparse matrix algebra [18], making this workload broadly representative of that class of algorithms.

Cryptosorter itself has a parametric memory system and can instantiate several parallel, banked memory interfaces to improve memory bandwidth. This approach works well for small numbers of sorting engines and banks, but if scaled to an extreme, results in large memory network queuing delays and degraded overall system

Table 1: FPGA resource utilization for memory system components.

Primitive	Slice Registers	Slice LUTS	18K-bit BRAM
Memory Ringstop	876	945	0
Memory Interleaver	1575	1766	0
Unified Central Cache	14499	16513	18
Single Central Cache	13195	15376	18
DRAM Controller	8525	13661	0
Private Memory Client	1660	2010	4
Coherent Memory Client	2985	5721	7
Coherence Controller	6795	7658	19

performance. We examine two configurations: a baseline configuration with a single private memory and a configuration with two memory interfaces, which we refer to as *banked*.

Filter: An HLS kernel that implements a filtering algorithm for K-means clustering [19]. K-means clustering partitions a data set of points into K clusters, such that each point belongs to the cluster with the nearest mean. *Filter* first builds a binary tree structure from the input data set and then traverses the tree in several iterations. Our implementation splits the tree into eight independently-processed sub-trees. Each partition tracks its tree traversal using a stack and maintains several sets of candidates for the best cluster centers. *Filter* uses 24 private memories: eight each for the sub-trees, stacks, and candidate center sets. Unlike the other applications, *Filter's* performance is sensitive to the latency of memory read responses. The workload chases data-dependent pointers and thus has limited ability to produce multiple, parallel memory requests.

6.1 Basic Memory Behavior

To build intuition about the behavior of our optimized memory systems, we benchmark three memory system implementations – a baseline implementation and two different interleaved implementations – using a memory performance kernel that varies both the working set size and reference locality (see Figure 8). In regions of high locality, when the working set is small, all three implementations have similar performance, since there are few misses to the backing memory system. As locality decreases and the number of accesses to the backing memory increases, our memory-interleaved implementations begin to outperform the baseline implementation. When all accesses are serviced in the central caches, our interleaved memory subsystems outperform the baseline throughput by 40% due to the availability of bandwidth from both DRAM controllers.

Our direct address interleaving mechanism routes requests based on a granularity related to the central cache line size. For *memperf* strides that are multiples of this granularity, our direct scheme will route all requests to a single memory controller, reverting to baseline performance. To combat this case, we introduce address hashing which recovers most of the DRAM bandwidth by evenly balancing the interleaving.

6.2 Area Consumption

Table 1 describes the area requirements of various components of the LEAP memory hierarchy. We consider two central cache controller implementations: the unified cache controller, which services two DRAM banks, and the single cache controller, which services a single DRAM. Managing a second DRAM bank marginally increases the area utilization of the cache controller, due to the increasing width of buses within the controller.

Within the memory system, the chief consumers of area are DRAM controllers and central cache controllers. Private and coherent memory clients require much fewer resources than the lower levels of the memory hierarchy. For most applications, the chief cost of implementing our bandwidth partitioning scheme is the introduction of a second central cache controller. Considered at the chip level, this area represents only 3.6% of the overall area of the VC709, which we believe is a small price to pay for the performance gains we will describe in subsequent sections.

Table 2: FPGA resource utilization for baseline and best performing memory configurations. In general, our optimizations increase utilization by about 20,000 slices and registers over the baseline. This represents 5.27% of LUTs and 2.44% of registers on the VC709.

Benchmark		Slice Registers	Slice LUTS	18K-bit BRAM
Memperf	Baseline	62317	80722	162
	Optimized	84166	105308	184
Cryptosorter (4 sorters)	Baseline	99284	137314	304
	Optimized	110116	146592	326
Heat (8-bit Data)	Baseline	157361	229276	243
	Optimized	189684	264881	284
Filter	Baseline	158774	185742	457
	Optimized	178039	207333	475
Average Utilization Increase		21067	22765	25.75
VC709 Area (%)		2.44%	5.27%	0.88%

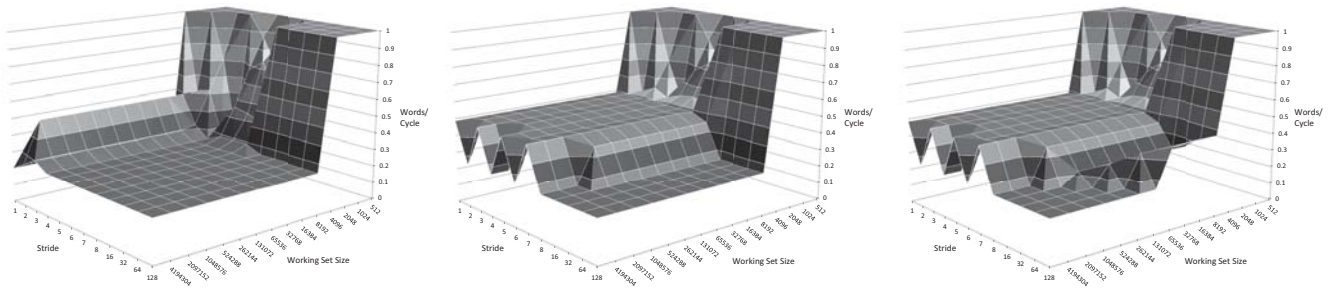
The chief overhead of our intelligent network synthesis is the introduction into the memory network of new ringstops. In the case where we simply assign memory clients to different memory networks without interleaving, no new hardware is introduced. Memory network ringstops that are capable of address interleaving, shown in Figure 6, are more than twice the cost of baseline ringstops, since interleaved ringstops must communicate with two or more controller networks. However, the cost of ringstops, and of the network in general, is dwarfed by the cost of implementing the other elements of the memory system: memory controllers and caches.

Table 2 shows the area utilization of baseline and best-performing implementations of each of our benchmarks. As expected, LMC optimization increases the area of each benchmark by approximately the area of a single cache controller. The largest increase occurs in the *heat* benchmark. In addition to a second central cache controller, the optimal instance of *heat* also includes a second coherence controller. However, the average increase in utilization is small relative to the size of the VC709: LUT utilization increases by 5.3% of the full VC709 while register utilization increases by 2.4%.

6.3 Randomized Partitioning

As a baseline for LMC optimization, we examine a random partitioning algorithm, in which memory clients are allocated to memory controllers in a randomized fashion. This balances the number of clients accessing each board-level memory, but is otherwise suboptimal. Figures 9, 10, and 11 show the relative performance of three benchmarks under randomized allocation. In general, random allocation is successful in improving program performance, especially for symmetric applications like *cryptosorter*. However, for throughput-oriented applications with asymmetric memory clients like *heat*, random partitioning gives only limited performance gains.

Like *heat*, *filter* also features asymmetric memory clients. However, random allocation actually performs slightly *better* than more sophisticated allocation schemes. This is because *filter* can sustain only a few outstanding requests per memory client, and is therefore less sensitive to bandwidth balancing. *Filter* is, however, extremely sensitive to latency. Random allocation both halves and balances



(a) Baseline (b) Address Interleaving (c) Hashed Address Interleaving

Figure 8: Performance of various LEAP memory systems. Performance is divided into two regions for each configuration. In the high-locality region, throughput is one word per cycle. In the low-locality region, performance is constrained to the bandwidth of the backing memory system. Our interleaving techniques nearly double the bandwidth over the baseline. Of note is the small region of increased bandwidth in the baseline memory system, which is due to the FPGA-side on-chip caching of DRAM lines.

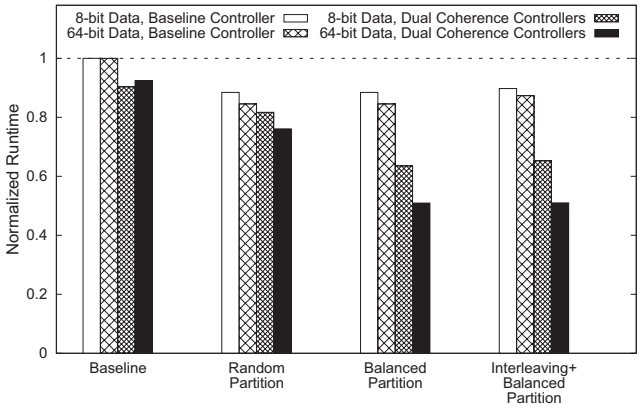


Figure 9: Runtime of *heat* on a 1-mega-entry array with various memory network configurations, normalized to the baseline implementations. *Heat* achieves the highest performance level when coherent and private cache network optimizations are composed.

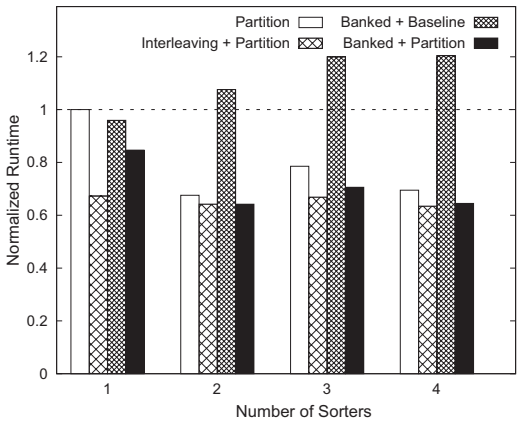


Figure 10: Runtime of *cryptosorter* on 256 kilo-entry lists with various memory network configurations, normalized to the baseline implementations. Generally, *cryptosorter* benefits from additional memory bandwidth. Banked versions of *cryptosorter* add latency and promulgate queuing delay in the memory network, lowering performance for large numbers of sorters.

memory network latency relative to the baseline, and, as a result, improves the performance of *filter* by about 15%.

6.4 Load-balanced Partitioning

The chief weakness of random partitioning is that it can sometimes oversubscribe the bandwidth of a single memory interface, especially when the memory clients are asymmetric in their memory bandwidth utilization. To further improve the system performance, we introduce load balancing. Load-balanced partitioning solves the problems of bandwidth imbalance by spreading memory accesses evenly across all board-level caches. *Heat*, which obtained some performance gains with random partitioning, obtains another 20% performance gain with load balancing, since heavily-loaded clients are evenly spread across the two board-level memories of the VC709. Load balancing naturally preserves the performance of symmetric applications like *cryptosorter*.

Load balancing evens out bandwidth, but ignores latency: low-bandwidth memory clients may all be assigned to the same network. In the case of *filter*, the memory network latencies are slightly imbalanced due to bandwidth balancing, which results in a small performance degradation compared to random partitioning.

6.5 Private Memory Address Interleaving

Load-balanced partitioning provides large performance gains for most of our benchmark cases. However, load balancing can result in small bandwidth imbalances if the bandwidth characteristics of a particular workload are uneven, or if the number of memory clients is relatively prime to the number of board-level memories. This bandwidth imbalance can lead to suboptimal performance. For example, in Figure 10, a three-sorter instantiation of *cryptosorter* experiences some benefit under load-balancing, but achieves less of a performance gain than either the two or four sorter case. Two of the sorters in the three sorter case must share a single controller. If we introduce memory interleaving, the odd controller's accesses can be spread across both controllers equally, leading to further performance gains. Similarly, applications like *memperf* or a single-sorter *cryptosorter*, which have only one client, can benefit from multiple controllers when using our interleaving approach.

Throughput-oriented applications generally benefit, or at least maintain load-balanced performance, with address interleaving. However, for latency-oriented applications, like *filter*, the extra cycles of latency added to route requests between memory networks result in a performance degradation, even as compared to our baseline implementation. This is the only case which we found in one of our optimizations failed to outperform the baseline, and suggests that care must be taken by the compiler when applying network optimizations to latency-oriented applications.

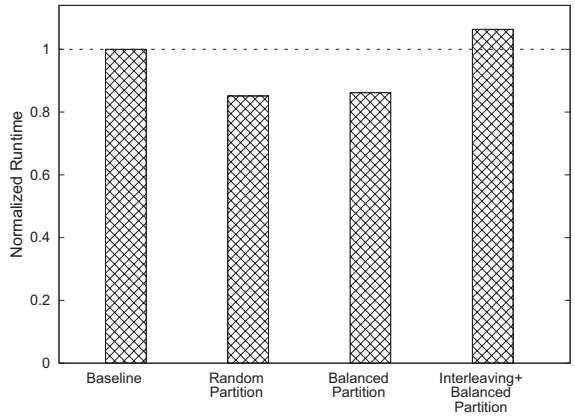


Figure 11: Runtime of *filter* with various memory network configurations, normalized to the baseline. *Filter* is sensitive to latency and thus benefits from random and load-balanced partitioning both of which reduce memory latency.

6.6 Coherent Cache Network Partitioning

Since LEAP coherent memories make use of private memories for intermediate data storage, they can take advantage of our bandwidth allocation and interleaving techniques in addition to our shared-memory-specific optimization. Figure 9 examines the *heat* benchmark under a variety of optimization scenarios.

Because most memory clients in *heat* are LEAP coherent memories, only the coherence controller in *heat* can utilize our private memory network optimizations. As a result, the performance gains for *heat* under our private memory optimizations are limited to about 12% in the best, load-balanced configuration.

Memory interleaving of the coherence domain at the coherence controller level provides a similar performance gain to load balancing, around 10%. However, because coherence controllers use private memory clients, we can compose the coherent memory interleaving technique with private memory optimizations. This composition of optimizations yields a performance gain of 49% for the 64-bit version of the *heat* benchmark and a 36% gain for the 8-bit version. The composed performance gain is actually better than the sum of the individual optimizations. This occurs because coherent memory interleaving introduces new coherence controllers and, thereby, increases the number of private memory clients, improving the effectiveness of our private memory optimizations.

6.7 Multiple FPGAs

Looking forward to cloud deployments [25] comprised of networks of FPGAs, we examine what happens when we stretch our synthesized memory networks between FPGAs. LEAP's named channel semantic permits their implementation as either inter-FPGA or intra-FPGA channels, differing only in latency. The LEAP compiler thus enables programs located on one FPGA to take advantage of potentially unused resources located on a nearby FPGA by automatically constructing a network between the FPGAs. The results of this experiment are shown in Figure 12.

Since inter-FPGA networks add latency, we examine only the throughput-oriented benchmarks: *heat* and *cryptosorter*. On the VC707, which has one DRAM per FPGA, the performance when scaling to two VC707s approaches that of the dual-DRAM VC709 for *heat*. *Cryptosorter* enjoys even larger performance gains when deployed to dual VC707s, and even obtains slightly better performance than a single VC709, indicating that the bandwidth offered by the remote memory outweighs the latency cost of accessing the

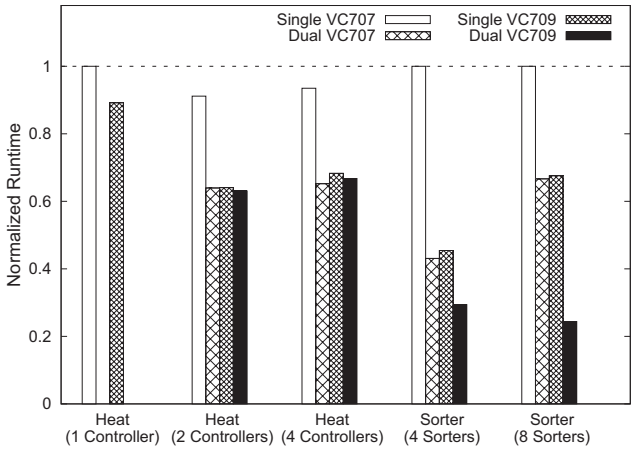


Figure 12: Normalized runtime of best-achieved performance solutions for 8-bit *heat* and *cryptosorter* with various platform configurations. *Heat* results are normalized to the runtime of the solution with a single coherence controller on a single VC707. The results of single-controller dual-FPGA solutions are not included because they simply add latency. *Cryptosorter* results are normalized to the runtime of the optimized single-VC707 solution.

remote memory. We note that it is unrealistic to expect designers to consider such complex systems without the assistance of a compiler to manage communication and resource allocation.

Scaling *cryptosorter* across two VC709s and to four memory banks yields another significant performance improvement. This is particularly pronounced for the eight sorter case, which shows superlinear performance gains: the baseline, single private memory controller case has a large memory network and suffers queuing delay. *Heat* also shows performance improvement when scaled to two FPGAs, but only about 2% over an optimized single VC709 implementation. Although incorporating multiple FPGAs exposes more memory resources, in the case of *heat*, increased bandwidth is counter-balanced by communication latency in the coherency networks. The effect of latency is also visible, to a lesser degree, in the slight performance degradation that occurs when we increase the number of interleaved coherence controllers. Our experience with the memory system of *heat* points to the need for better program analysis when scaling application-specific memory systems, especially if large performance cliffs, like inter-chip latency, are present in the scaled system.

Our results show, particularly for throughput-oriented applications, that bandwidth borrowing or sharing among adjacent, networked FPGAs can be a significant source of performance gains.

7. CONCLUSION

Modern FPGA boards include multiple memory resources to support the increased bandwidth demand of large numbers of computational resources. To alleviate the complexity of designing programs for such systems, we have demonstrated a resource- and application-aware compiler, the LEAP Memory Compiler, that can transparently optimize the memory system of a given application. Using runtime statistics, LMC automatically partitions the network that connects user-specified memory interfaces to board-level memory resources, simultaneously increasing the memory bandwidth and reducing memory latency. We target several platforms wherein multiple on-board memory resources are available, and demonstrate that LMC produces significant performance gains.

To optimize private memory interfaces, we balance the total traffic

on each of several partitioned memory networks so that asymmetric memory clients can efficiently share the memory bandwidth. For memory clients that consume large amounts of bandwidth, a compiler-synthesized memory interleaver may be added to partition a single client's address space into multiple disjoint regions. A similar memory interleaving technique is also applied to shared memory interfaces. We also construct distributed coherence controllers to relieve network contention and introduce hierarchical coherence rings to improve the latency of coherent cache networks. Our evaluation shows that the compiler-synthesized cache network provides up to 49% performance gains to throughput-oriented workloads we studied and a 15% performance gain to the latency-oriented workload we studied, while increasing design area utilization by less than 6%. We also demonstrate that FPGA applications can make use of and benefit from remote memory resources accessible by an inter-FPGA network, and that throughput-oriented applications can derive significant performance gains as a result.

In this work, we show that different optimizations are helpful for different applications. One direction for future work is to explore more non-uniform workloads and conduct a more detailed run-time analysis on clients' bandwidth and latency demands to better direct compiler optimizations. For example, such analysis can be used to classify different memory clients within a single application as throughput-sensitive or latency-sensitive, enabling the compiler to synthesize a more complicated network topology. A latency-oriented client would be given a more direct connection to a memory controller, while a throughput-oriented client could be given some quality-of-service guarantees ensuring higher aggregate bandwidth. In addition, we also plan to explore dynamic partitioning for memory clients whose memory demands change over time or between executions.

8. REFERENCES

[1] Achronix semiconductor corp. *http://www.achronix.com/*.

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

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

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

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[6] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *FCCM*, 2012.

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

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[9] G. Dessouky, M. Klaiber, D. Bailey, and S. Simon. Adaptive dynamic on-chip memory management for FPGA-based reconfigurable architectures. In *FPL*, 2014.

[10] K. Farkas, Z. Vranesic, and M. Stumm. Cache consistency in hierarchical-ring-based multiprocessors. In *IEEE Conference on Supercomputing*, 1992.

[11] K. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. S. Emer. Leveraging latency-insensitivity to ease multiple FPGA design. In *FPGA*, 2012.

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

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

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

[15] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Hawaii International Conference on System Sciences*, 1994.

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

[17] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[18] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[19] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

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

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

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

[23] 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.

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

[25] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

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

[27] F. Winterstein, S. Bayliss, and G. A. Constantinides. Separation logic-assisted code transformations for efficient high-level synthesis. In *FCCM*, 2014.

[28] F. Winterstein, K. Fleming, H. Yang, S. Bayliss, and G. A. Constantinides. MATCHUP: memory abstractions for heap manipulating programs. In *FPGA*, 2015.

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