Automatic Application-Specific Optimizations under FPGA Memory Abstractions
by
Hsin-Jung Yang
B.S., National Taiwan University (2010)
S.M., Massachusetts Institute of Technology (2012)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
June 2017
© Massachusetts Institute of Technology 2017
All Rights Reserved.

Signature redacted
Department of Electrical Engineering and Computer Science
May 12, 2017

Signature redacted
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted
Joel Emer
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted
A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair of the Department Committee on Graduate Students
Automatic Application-Specific Optimizations under FPGA Memory

Abstractions

by

Hsin-Jung Yang

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 2017, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

FPGA-based accelerators have great potential to achieve better performance and energy-efficiency compared to general-purpose solutions because FPGAs permit the tailoring of hardware to a particular application. This hardware malleability extends to FPGA memory systems: unlike conventional processors, in which the memory system is fixed at design time, cache algorithms and network topologies of FPGA memory hierarchies may all be tuned to improve application performance. As FPGAs have grown in size and capacity, FPGA physical memories have become 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 automated build procedures that can make good use of memory resources by performing resource-aware, application-specific optimizations.

In this thesis, we leverage the freedom of abstraction to build program-optimized memory hierarchies on behalf of the user, making FPGA programming easier and more efficient. To enable better generation of these memory hierarchies, we first provide a set of easy-to-use memory abstractions and perform several optimization mechanisms under the abstractions to construct various memory building blocks with different performance and cost tradeoffs. Then, we introduce a program introspection mechanism to analyze the runtime memory access characteristics of a given application. Finally, we propose a feedback-directed memory compiler that automatically synthesizes customized memory hierarchies tailored for different FPGA applications and platforms, enabling user programs to take advantage of the increasing memory capabilities of modern FPGAs.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Joel Emer
Title: Professor of Electrical Engineering and Computer Science
Acknowledgments

I would like to acknowledge the support of many people who made this thesis possible. First and foremost, I would like to express my sincere gratitude to my advisors, Professor Srinivas Devadas and Professor Joel Emer, for their encouragement and support throughout these years. Srini's enthusiasm for research is contagious and I am deeply grateful for the freedom he has given me to pursue the research that interests me most. I especially want to thank Joel, whom I have worked closely with through my PhD study. Being a great mentor and role model, Joel has always been patient and supportive. I appreciate the invaluable research guidance and thoughtful career advice he has given me as well as all the life stories he has shared with me. I would also like to thank my thesis committee member, Professor Daniel Sanchez, for providing insightful feedback and constructive suggestions to improve the quality of this thesis.

I want to extend my gratitude to my collaborators, Elliott Fleming and Michael Adler from Intel, Felix Winterstein from Imperial College London, and Annie Chen from MIT. Working with them has been a pleasant experience. Without them, I would not be able to complete my thesis projects. I would like to profoundly thank Elliott and Michael for their immeasurable help in brainstorming research ideas, conducting evaluation experiments, and writing research papers. Through collaboration and weekly phone meetings with them (as well as countless Skype conversations with Elliott), I have learned a lot in the field of reconfigurable computing and improved my paper writing and presentation skills. I am also grateful for the successful and productive collaboration with Felix. Felix have developed many high-level synthesis applications that are important benchmarks for performance evaluation. I want to thank Annie, who has also been a good friend of mine for 14 years, for helping me formulate optimization problems and develop algorithms presented in the thesis.

I am thankful to the members of the Hornet group for creating a great working environment. I appreciate the stimulating conversations and helpful discussions with them. Special thanks to Ilia Lebedev, who has helped me fix my FPGA evaluation board so I could run the final set of experiments.
Many thanks go to all of my friends who have supported me through the ups and downs of my PhD journey and made my time at MIT enjoyable. Thanks Amy Chen for providing a lot of help in research and life, including sharing her beautifully formatted thesis template with me. Thanks Lixin Sun and Annie Chen for sharing wonderful times with me as well as their tasty homemade meals and desserts. I would also like to thank Chern Chuang for encouraging me, cheering me up, and always being there for me during the past three years. Finally, I am eternally grateful to my parents for their unconditional love and full support throughout my life.
Abstract

Acknowledgments

Contents

List of Figures ........................................ xi
List of Tables ........................................ xv
List of Algorithms .................................... xvii

1 Introduction

1.1 Thesis Contribution ................................. 7
1.2 Thesis Organization ......................... 9

2 Background

2.1 The LEAP FPGA Operating System ................. 11
  2.1.1 Model of Computation .......................... 12
  2.1.2 LEAP Private Memory ......................... 15
  2.1.3 LEAP Compilation ............................ 17
2.2 Extending LEAP Memories to HLS Applications .... 18
2.3 FPGA Platforms and Workloads for Evaluation .... 20
  2.3.1 Hand-Assembled Applications ................. 21
2.3.2 HLS-Compiled Applications ........................................... 24
2.3.3 FPGA Platforms ......................................................... 26

3 Related Work ................................................................. 29
3.1 FPGA High-Level Programming Environments ....................... 29
3.2 FPGA High-Level Synthesis Tools ..................................... 30
3.3 FPGA Memory Abstractions ............................................. 32
3.4 Managing Coherency and Synchronization on FPGA .............. 34
3.5 Memory Optimizations .................................................... 35
  3.5.1 Memory Optimizations in Processors ............................... 36
  3.5.2 Memory Network Optimizations in Embedded SoCs ........... 39
  3.5.3 Memory Optimizations on FPGAs ................................ 40

4 Shared Memory Abstraction ................................................. 43
4.1 LEAP Coherent Memory .................................................. 45
  4.1.1 Coherent Memory Interface ........................................ 46
  4.1.2 Coherence Protocol .................................................. 47
  4.1.3 Coherent Memory Architecture .................................... 49
  4.1.4 Coherent Memory Client Microarchitecture ..................... 50
  4.1.5 Coherence Controller Microarchitecture ......................... 52
  4.1.6 Deadlock Freedom .................................................... 53
  4.1.7 Memory Consistency ................................................ 54
4.2 Coherent Cache Network Optimization .................................. 55
4.3 Synchronization Primitives .............................................. 59
  4.3.1 Lock Service ......................................................... 59
  4.3.2 Barrier Service ...................................................... 60
4.4 Evaluation ..................................................................... 62
  4.4.1 Coherent Memory Service .......................................... 62
  4.4.2 Synchronization Service ............................................. 66
  4.4.3 Implementation Area ................................................ 67
4.5 Summary ...................................................................... 68
5 Cache Optimizations

5.1 Cache Prefetching ........................................ 72
  5.1.1 Prefetching In FPGAs ................................. 72
  5.1.2 Prefetching Microarchitecture ......................... 74
  5.1.3 Evaluation ........................................... 78

5.2 Scalable Caches ........................................... 83
  5.2.1 On-chip Shared Cache ................................. 84
  5.2.2 Cache Scalability ................................... 87
  5.2.3 Non-power-of-two Caches ............................. 88
  5.2.4 Design Tradeoff ..................................... 90
  5.2.5 Compile Time Optimization ............................ 91
  5.2.6 Evaluation ........................................... 93

5.3 Summary .................................................. 105

6 Service Communication Abstraction ......................... 107

6.1 Motivating Example ....................................... 108

6.2 Communication Abstraction ............................... 109

6.3 Network Topologies ....................................... 111

7 LEAP Memory Compiler ..................................... 115

7.1 Compiler Overview ....................................... 117

7.2 Construction of Optimized Cache Networks .............. 120
  7.2.1 Resource-Aware Program-Optimized Memory Partitioning 121
  7.2.2 Cache Network Profiler ................................ 125
  7.2.3 Program-Optimized Tree Networks ..................... 128

7.3 Evaluation ............................................... 132
  7.3.1 Cache Network Partitioning ........................... 133
  7.3.2 Program-Optimized Cache Networks ................... 142

7.4 Summary .................................................. 150
8 Conclusion

8.1 Thesis Summary ........................................ 153
8.2 Future Work ............................................ 156

Bibliography ........................................... 159
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>An example of a program-optimized FPGA memory hierarchy.</td>
<td>8</td>
</tr>
<tr>
<td>2-1</td>
<td>A pair of modules connected by a latency-insensitive channel. The send and</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>receive endpoints with the same channel name (&quot;AtoB&quot;) are matched during</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compilation.</td>
<td></td>
</tr>
<tr>
<td>2-2</td>
<td>LEAP private memory interface.</td>
<td>15</td>
</tr>
<tr>
<td>2-3</td>
<td>An example of LEAP private memory hierarchy.</td>
<td>16</td>
</tr>
<tr>
<td>2-4</td>
<td>LEAP compilation flow.</td>
<td>17</td>
</tr>
<tr>
<td>2-5</td>
<td>Automating the integration of HLS applications and LEAP-generated platforms.</td>
<td>20</td>
</tr>
<tr>
<td>2-6</td>
<td>A simple nested loop that models heat transfer from time 0 to time T on an</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>$M \times N$ grid. $C_0, C_r, C_g$ are constants related to thermal diffusivity.</td>
<td></td>
</tr>
<tr>
<td>4-1</td>
<td>The LEAP coherent memory interface extended from the private memory</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>interface with extensions highlighted.</td>
<td></td>
</tr>
<tr>
<td>4-2</td>
<td>Coherent memory architecture.</td>
<td>49</td>
</tr>
<tr>
<td>4-3</td>
<td>Coherent cache microarchitecture.</td>
<td>51</td>
</tr>
<tr>
<td>4-4</td>
<td>Coherence controller microarchitecture.</td>
<td>52</td>
</tr>
<tr>
<td>4-5</td>
<td>LEAP coherent memories with dual coherence controllers.</td>
<td>55</td>
</tr>
</tbody>
</table>
4-6 A walkthrough example of coherency management in a multi-controller system. ......................................................... 57
4-7 A lock service walkthrough example. ............................................. 61
4-8 LEAP memory hit latency comparison. ........................................ 63
4-9 LEAP memory throughput comparison. ....................................... 63
4-10 Performance comparison of different coherent memory configurations for heat with a various number of worker engines. .............. 64
4-11 Performance comparison of coherent memories with single and dual coherence controllers when running heat at various frame sizes with 16 worker engines. ............................................. 64

5-1 The microarchitecture of the LEAP private memory client's private cache with prefetching logic. Our augmentations are highlighted. .... 74
5-2 Prefetcher performance results normalized to a non-prefetched implementa-tion. ............................................................... 80
5-3 Prefetch result analysis for MMM. ............................................. 81
5-4 The extended LEAP private memory hierarchy, including scalable BRAM caches and a shared on-chip cache. Our modifications are highlighted. 85
5-5 Set associative cache microarchitecture. ..................................... 86
5-6 BRAM store implementation option. .......................................... 88
5-7 Cache indexing mechanisms for power-of-two and non-power-of-two caches. 89
5-8 An example of a LEAP program built with the two-phase compilation flow. 92
5-9 L1 cache performance with various microarchitectures. ................. 95
5-10 Performance metrics for HAsim with various cache configurations. ... 97
5-11 Performance comparison for heat with various cache configurations. . 99
5-12 Performance comparison for HLS kernels with various L1 cache sizes.
Results are normalized to the performance of the smallest monolithic cache. 100
5-13 Performance comparison of the banked cache against the monolithic cache for various benchmarks built with maximal L1 cache sizes. 101
5-14 Implementation space of L2 caches. ......................................... 101
5-15 Relative throughput gain for memperf: ........................................ 102
6-1 The abstract interfaces of service connections. .............................. 110
6-2 Communication abstraction for centralized services. .................... 111
6-3 Examples of compiler-generated network topologies. ..................... 112
7-1 Extended LEAP compilation flow with our extensions highlighted in blue. 118
7-2 Program instrumentation built with the LEAP statistics collection service. 119
7-3 LEAP private memory system with a partitioned cache network. ........ 121
7-4 LEAP private memory interleaver logic microarchitecture. ............... 124
7-5 LEAP dynamic parameter service. .............................................. 126
7-6 An application-specific network profiler with instrumentation logic and latency FIFOs inserted at each memory client. ......................... 127
7-7 Performance of memperf for various LEAP memory systems. ........... 134
7-8 Performance of mergesort on 256 kilo-entry lists with different memory network configurations and a various number of parallel sorters, averaged over all the sorters and normalized to the baseline implementations. . . 137
7-9 Performance of beat on a 1-mega-entry array with various memory network configurations, normalized to the baseline implementations with a single coherence controller. ............................... 137
7-10 Performance of filter and reflect-tree with various memory network configurations, normalized to the baseline implementations. ............ 138
7-11 Performance of best-achieved performance solutions for 8-bit beat and mergesort with various platform configurations, normalized to single-VC707 solutions. ..................................................... 141
7-12 Latency sensitivity of memory clients in filter. .............................. 144
7-13 Simulated performance of filter with various tree construction algorithms. 145
7-14 Performance comparison of various network configurations for single-program applications, normalized to the actual baseline implementations. 146
7-15 Performance comparison of various network configurations for the mergesort-filter application. .................................................. 148
Performance comparison for the heat filter application.
List of Tables

2-1 FPGA platforms for evaluation. ........................................ 26

4-1 Shared queue test runtime result. ........................................ 66

4-2 Throughput comparison of barrier services. ............................. 66

4-3 FPGA resource utilization and maximum frequency of shared memory
primitives. ........................................................................... 67

4-4 Normalized area of coherent caches with various configurations. .... 68

5-1 Structural and performance metrics for board-level memories on FPGA
evaluation platforms, as measured at the memory controller. .......... 78

5-2 FPGA resource utilization and maximum frequency of prefetching logic. 82

5-3 Post-place-and-route results for applications with different cache sizes. . 94

5-4 L2 cache performance gain for merger. ...................................... 103

5-5 Power and energy measurements for HLS applications†. .............. 104

7-1 FPGA resource utilization for memory system components. .......... 135

7-2 Resource utilization for baseline and best performing memory configura-
tions. ............................................................................... 136

7-3 Resource utilization for various network primitives. ..................... 143

7-4 Resource utilization for the cache network in filter. ..................... 143
List of Algorithms

6-1 Arbiter with bandwidth control. ........................................ 113
7-1 Private cache network partitioning. ..................................... 123
7-2 Construct a minimum weight tree using DP. ..................... 131
Field-programmable gate arrays (FPGAs), which were originally used for application-specific integrated circuit (ASIC) prototyping and emulation, have grown in capacity and become increasingly popular as accelerators in both academia and industry [15, 34, 73, 77, 78, 81, 90]. Massive parallelism and specialization make FPGAs attractive to improve the performance and power efficiency of applications that conventionally run on general-purpose machines [36, 55, 89]. To achieve high performance and efficiency, FPGA programmers have traditionally utilized low-level primitives and hardware description languages (HDLs), such as VHDL and Verilog, manually managing fine-grained parallelism and 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. The difficulty of designing hardware at register transfer level (RTL) has also inhibited the wide adoption of FPGA-based solutions.

To reduce programming effort and shorten development time, recent work has focused on raising the level of abstraction available to FPGA programmers. Several higher-level programming languages, such as Bluespec [12], Lime [8], and Chisel [9], as well as associated compiler frameworks that automate the transformation from these languages to RTL implementations have been proposed. Numerous C-to-hardware compilers, including Xilinx Vivado High-Level Synthesis [94], ROCCC [93], LegUp [13], Catapult C [14], Impulse C [47], and Cadence Stratus High-Level Synthesis [86], have been
developed to extract parallelism from C/C++/SystemC source codes and automatically map software kernels to RTL specifications, making FPGA programming easier and more acceptable for software programmers.

In addition to these high-level synthesis (HLS) tools, another class of work has added new programming primitives to existing HDLs, enabling HDL programmers to concisely express program behavior at a higher level than register transfers. Examples of higher-level constructs include communication services, to software [57, 85], within the FPGA [76], and between FPGAs [32], as well as memory abstractions [3, 19, 57]. These primitives, which abstract away vendor-specific interfaces and resources, can be configured and reused for various applications across different FPGA platforms.

High-level abstractions provide clearly-defined, generic interfaces that separate the user program from underlying infrastructure implementations, such as memory, communication, and other ancillary services. FPGA programmers can therefore concentrate on algorithm design while programming against fixed interface layers. These interface layers allow users to write portable programs, which can run on different platforms without changes to the application code. Underneath the interfaces, low-level platform implementation details are handled by a combination of compilers and system developers. Such abstractions enable the platform to be optimized separately, improving program performance without perturbing the original user design. However, there is rarely a globally optimal platform design strategy for different applications. High-level abstractions and productivity may come at the expense of 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 develop algorithms and tools that optimize platform service implementations in an application-specific manner on behalf of programmers.

This thesis primarily focuses on one aspect of platform optimizations: FPGA memory systems. In both conventional processors and FPGAs, the memory system is critical to overall program performance for a broad class of applications. The memory system on a specific conventional processor is fixed and therefore a compromise based on a set of expected workloads to achieve the best average performance. In contrast, cache
algorithms or the memory hierarchies on FPGAs can be tailored for different applications. For example, a well-pipelined, bandwidth-intensive program might not need a cache at all, while a latency-sensitive program, like a soft-processor or a graph algorithm implementation, might prefer a fast first-level cache. Building customized memory systems to achieve high program performance involves exploring a large design space. Therefore, it is important to have an automated build procedure that facilitates this exploration process.

In addition, FPGA programs rarely consume all the resources available on a given FPGA, partly due to design difficulty such as cache scalability and partly due to design reuse. Existing platform abstractions enable designs to be ported across FPGAs, but a large amount of resources may be left unused when an algorithm design targeting an older, smaller FPGA is ported to a newer, larger FPGA. Producing efficient and portable FPGA designs requires resource-aware platform optimizations that exploit the unused resources to improve performance. For example, a resource-aware memory compiler might utilize spare resources to build prefetching logic or to increase the size of the on-chip caches.

Automatic, resource-aware optimizations are especially important for utilizing coarse-grained resources, like board-level memories, which have traditionally been difficult to integrate at design time. As the availability of more transistors makes it feasible to build larger, bandwidth-hungry designs as well as the memory controllers necessary to feed them, modern FPGA boards have begun to include multiple dynamic random-access memories (DRAMs) [68, 104]. Moreover, the number of memory controllers is increasing rapidly as vendors move to harden memory interfaces [2, 68]. To improve performance, it is critical for a memory compiler to make good use of these increasingly rich and varied memory resources without drastically increasing the design burden.

This thesis aims to automatically construct an optimized memory system tailored for a given FPGA application on a particular platform. To achieve this goal, the key requirements are memory abstractions, optimizations performed under the abstraction, and an automated build procedure that selects and implements the best optimizations. First, we need a set of easy-to-use memory abstractions to support a wide variety of applications, including programs written in RTL and in high-level languages (such as
Chapter 1. Introduction

C/C++ or domain-specific languages). These memory abstractions separate the user program from details of the memory system implementation, thus enabling changes in the memory system without requiring modifications in the user code. Second, we need various optimization mechanisms that are performed under the abstraction to construct a rich set of memory building blocks with different performance and cost tradeoffs. Finally, we need a build procedure that automatically composes an application-specific hierarchy from these memory building blocks. This build procedure includes program introspection, which characterizes the program’s runtime memory access behavior. The introspection results, together with program resource utilization, are used by a feedback-directed compiler to guide the selection of optimal memory building blocks and the implementation of an optimized memory hierarchy tailored for the target application and platform.

Recent research has provided a number of abstract memory interfaces that hide memory implementation details from application designers and enable compilers to assist in the construction of application-specific memory systems. For example, the Connected RAM (CoRAM) [19] architecture defines an application environment that separates computation from memory management. CoRAM memory accesses are managed by control threads that fetch data from off-chip memories to on-chip buffers using a C-like language. LEAP (Latency-insensitive Environment for Application Programming) private memories [3] provide a memory abstraction with a simple request-response-based interface and manage a memory hierarchy extending from on-chip block RAMs (BRAMs) to the host processor’s memory. We adopt the LEAP private memory abstraction as the base of this thesis work, because it provides a simple user interface, while giving us enough flexibility to construct a diverse set of memory hierarchies.

To make application development easier for FPGA programmers, it is important to support programs written in higher-level languages. In addition to RTL-based, hand-assembled applications, which can be easily integrated with the baseline LEAP memories, we extend the memory services to programs that are written in C/C++ and compiled through C-to-hardware HLS tools, allowing these applications to efficiently make use of available memory resources. These HLS tools generally have limited support for
system-level integration as well as fine-grained memory management and optimizations for on-board memories. Taking advantage of the LEAP abstract memory interface, we provide HLS applications with our easy-to-use, optimized memory service by connecting the memory systems through standard bus protocols.

To support a wider range of applications, we extend the scope of the basic LEAP private memory abstraction by proposing a shared memory abstraction to facilitate mapping parallel algorithms to FPGA, which has been considered to be difficult due to limited infrastructure support. Without good shared memory libraries, programmers usually develop FPGA-based parallel algorithms by adopting a distributed (non-shared) memory model and explicitly handling all data sharing between processing engines [44, 83]. This approach prolongs development time because programmers are fully exposed to the complexity of distributed coordination. In order to simplify parallel programming on FPGA, we propose a set of declarative primitives that maintain coherency and consistency of accesses to shared memory resources. We design the LEAP coherent memory, which manages coherent caches under a simple request-response-based interface similar to that of LEAP private memories. We also provide lock and barrier primitives, which leverage the native communications primitives of the FPGA rather than relying on shared memory. Because the proposed primitives are framed in terms of high-level communication channels, they can be automatically partitioned across any configuration of FPGAs.

To enable better generation of program-optimized memory systems, we perform several optimizations under the private and shared memory abstractions to construct various memory building blocks with different latency, bandwidth, and frequency properties. These optimizations do not need to benefit all FPGA applications since they are used only when necessary, much in the same way as conventional compilers employ a particular optimization only when it improves the target application. We first examine automatically generated prefetching as a means of using spare resources left by the user program to improve performance. As in general-purpose processors, prefetchers can be introduced alongside the existing memory hierarchy without modifying the application. We present a novel, FPGA-optimized microarchitecture for prefetching that is tuned based on the behavior of typical FPGA applications.
To efficiently utilize on-chip memory resources, we design several new cache implementations, mostly targeting very large on-chip caches. We target large caches because application kernels may only explicitly use few memory resources and substantial memory capacity may be available to the platform for use on behalf of the user program. FPGA-based on-chip caches are built by aggregating distributed on-die BRAMs, which are typically assumed to have single-cycle access latency. As these caches scale across the chip, the wire delay between the BRAM resources increases, eventually causing operating frequency to drop and potentially decreasing program performance. To relieve timing pressure for building large BRAM caches, we provide microarchitectural techniques to trade memory latency for design frequency. We also study cache design tradeoffs among access latency, operating frequency, and cache capacity for FPGA applications with different memory access behavior.

Both prefetching and cache scaling focus on the microarchitecture of on-chip caches to improve cache performance, including the cache bandwidth and hit rate. In a multi-level memory hierarchy, the performance of the memory system may also be influenced by the on-chip cache network that connects different levels of caches to forward messages such as read misses and write-backs. Cache network customization is especially valuable for applications with multiple, asymmetric memory clients where individual clients have varying bandwidth and latency sensitivities. For example, a memory client with higher data locality or with a deeply pipelined computational engine may be able to tolerate longer cache network latency, while a memory client with lower data locality or lower request-level parallelism may benefit more from reduced network latency.

To facilitate the exploration of different cache network topologies, we introduce a new communication abstraction that provides a clean separation between the functionality of the cache network and physical topologies. This new communication abstraction, which merely defines the endpoint interfaces, allows a compiler to freely construct any network topologies, study design tradeoffs including network scalability and router complexity, and optimize the network for each application. This is analogous to the use of memory abstractions for performing application-specific memory optimizations.
1.1. Thesis Contribution

Finally, to improve productivity, we develop a build procedure that helps programmers construct application-specific memory systems from the available memory building blocks and configurable cache network. This is the final step for automating the construction of program-optimized memory hierarchies. We provide a framework that enables programmers to easily configure multiple levels of caches through parameters. In addition, we propose the LEAP Memory Compiler (LMC), which can automatically synthesize application-optimized memory hierarchies by enabling automatic, resource-aware optimizations through feedback-directed compilation. LMC operates in three phases: instrumentation, analysis, and synthesis, providing customized solutions for different applications and platforms. LMC optionally injects instrumentation infrastructure into the memory system to collect runtime information about the way the program uses memory. Then, LMC performs optimizations based on the available memory resources in the system, the number of FPGAs, as well as the user program’s memory behavior derived from runtime instrumentation.

1.1 Thesis Contribution

By providing a resource-aware memory compiler with several configurable memory abstractions and optimizations performed under these abstractions, this thesis demonstrates that it is feasible to automatically construct an FPGA-based memory system that is optimized for both the target application and the target platform on behalf of the programmer. The following list summarizes the contributions of this thesis:

- An extended framework that provides both HLS-compiled kernels and hand-written RTL applications with configurable memory service implementations.

- A shared memory abstraction that consists of coherency and synchronization primitives to coordinate accesses to the shared memory, facilitating the development of parallel algorithms on FPGAs. (This work was first published in [105] and then included in [35].)
Chapter 1. Introduction

Figure 1-1: An example of a program-optimized FPGA memory hierarchy.

- A configurable cache prefetcher that is tuned based on the behavior of typical FPGA applications as well as FPGA design tradeoffs between resource utilization and operating frequency. (This work was published in [106].)

- Microarchitectural techniques to build scalable on-chip caches that efficiently exploit spare memory resources left by application kernels. (This work was first published in [108] and its extension was published in [109].)

- A communication abstraction that separates the functionality of memory network from physical implementations. (This work was published in [110].)

- A feedback-driven memory compiler that automatically constructs a program-optimized memory hierarchy based on available resources and program introspection results. (This work was published in [107] and [110].)

- A demonstration of the effectiveness of the proposed memory optimizations through a set of FPGA platforms and workloads.
1.2 Thesis Organization

Figure 1-1 illustrates an example of a program-optimized FPGA memory hierarchy to highlight the thesis contributions. This memory hierarchy can be constructed using the abstractions, optimizations, configuration framework, and feedback-driven compilation flow proposed in this thesis. In this example, the FPGA is paired with a host processor. The user engines on the FPGA are computational kernels that operate on data backed by the host processor's memory. The data is cached on FPGA through a memory hierarchy, where the FPGA board-level memories are used as last-level caches. The user engines can be either hand-written RTL modules or software kernels compiled through HLS. For engines that need to operate on shared data, coherent caches and synchronization primitives can be instantiated to manage shared-memory accesses. Both private and coherent caches can be configured to include prefetching logic or be scaled to consume spare resources left by the user program. The private on-chip caches are connected to board-level memories through a compiler-optimized memory network, which efficiently utilizes the memory bandwidth offered by the board-level memories and minimizes the performance impact introduced by network latency.

1.2 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 begins with a discussion of prior work this thesis builds upon: specifically the LEAP FPGA operating system [35]. Chapter 2 also describes how we extend the support of memory services to HLS applications and introduces the collection of FPGA platforms and applications that we use to evaluate the effectiveness of the proposed optimizations. Chapter 3 discusses the rest of prior work related to this thesis, including other FPGA high-level programming models and abstractions as well as memory and network optimizations for processors, FPGAs, or systems on a chip (SoCs).

After the discussion of prior work, Chapter 4, 5, 6 and 7 cover the thesis contributions in detail. Chapter 4 describes the shared memory abstraction as well as the design of associated coherency and synchronization primitives. Chapter 5 introduces the proposed cache optimization techniques that can be applied to speed up target applications by
Chapter 1. Introduction

exploiting spare resources. Chapter 6 describes the communication abstraction that facilitates the examination of the cost-performance tradeoffs for different cache network topologies. Chapter 7 presents the details of LMC, including the compilation flow, program analysis mechanisms, and algorithms for selecting an optimal memory hierarchy. These chapters also discuss the performance of the proposed design and optimizations by examining the workloads described in Section 2.3. Finally, Chapter 8 concludes this thesis and provides some future research directions.
The goal of this thesis is to automatically construct program-optimized memory hierarchies for a wide range of applications. To achieve this goal, this thesis builds upon existing platform abstractions and a multi-FPGA compiler included in the LEAP FPGA operating system [35]. This chapter first outlines the LEAP operating system in Section 2.1. Section 2.2 describes how we improve usability of LEAP memories by extending the memory interfaces to support C-based application kernels compiled through HLS. Finally, Section 2.3 introduces the FPGA platforms and applications we use to evaluate the design and optimizations proposed in this thesis.

2.1 The LEAP FPGA Operating System

The LEAP FPGA operating system is an open-source framework\(^1\) that eases the burden of FPGA programming by offering a set of platform services, including standard I/O, memory management, and communication. LEAP also provides device abstraction layers with fixed interfaces for a number of commercially available FPGA platforms from both Xilinx and Altera (now part of Intel), enabling LEAP programs to be portable across different FPGAs. LEAP is written in Bluespec SystemVerilog [12], which provides a powerful static elaboration phase and some other features that simplify platform service implementations. The basic concepts of LEAP were first introduced in [75]. In [35], a

---

\(^1\)LEAP source codes and documentations are available on GitHub: http://www.leap-fpga.org.
Chapter 2. Background

later version of LEAP\(^1\) included various extensions: a multi-FPGA compiler, which can automatically partition the target design across multiple FPGAs, and the shared memory abstraction, which is a part of the contributions of this thesis.

For the base of this thesis work, we adopt LEAP private memories [3] as the basic memory abstraction. This memory primitive is built on top of LEAP's latency-insensitive channels [32], named communications primitives that are instantiated within user programs and implemented by the LEAP compiler [31].

### 2.1.1 Model of Computation

Traditional hardware programming describes designs in terms of registers, logic gates, wires, and clocks, explicitly controlling the cycle-level behavior of the hardware design. This cycle-accurate computational model and wire-based communication make altering platform service implementations difficult to maintain the overall functional correctness of the system without perturbing the user program. To decouple various platform services from the user program, LEAP builds upon a fundamental communication abstraction: latency-insensitive channels [32].

LEAP latency-insensitive channels provide named point-to-point communications for hardware programs. The interfaces and operating behaviors of latency-insensitive channels are similar to those of RTL FIFO modules: simple enqueue and dequeue operations with some status signals indicating whether the FIFOs are empty or full. However, unlike the traditional hardware FIFOs, which have a fixed implementation with explicitly specified buffering and latency, latency-insensitive channels make only the following basic guarantees: (i) Delivery of messages is guaranteed, i.e., the underlying network should be deadlock-free. (ii) Delivery is in-order. (iii) At least one message can be in flight at any point in time. This implies that a latency-insensitive channel may have dynamically-variable transport latency and arbitrary, but non-zero, buffering. Programmers who instantiate latency-insensitive channels need to make sure that the timing variations will not impact the functional correctness of their design.

---

\(^1\)LEAP stands for Logic-based Environment for Application Programming in [75] and stands for Latency-insensitive Environment for Application Programming in [35].
2.1. The LEAP FPGA Operating System

```verilog
module mkModuleA;
    Send#(Bit(36)) toB <- mkSend("AtoB");
endmodule

module mkModuleB;
    Recv#(Bit(36)) fromA <- mkRecv("AtoB");
endmodule
```

Figure 2-1: A pair of modules connected by a latency-insensitive channel. The send and receive endpoints with the same channel name ("AtoB") are matched during compilation.

LEAP adopts the syntax of Soft Connections [76] to describe latency-insensitive channels. Soft Connections provide an extension to the base Bluespec syntax, allowing programmers to specify a logical topology of communication that is separated from the physical implementation. Soft Connections originally supported named latency-insensitive point-to-point channels for designs on a single FPGA. The network endpoints are automatically connected during static elaboration. The LEAP compiler [31] extends the concept of Soft Connections to automatically construct inter-FPGA networks.

Figure 2-1 shows an example of two modules connected by a latency-insensitive channel, which has a pair of named send and receive endpoints. At compilation time, send and receive endpoints in the user program are matched, and a flow-controlled channel implementation is instantiated between them. With the abstract interface, the compiler is free to choose any channel implementations from simple fixed-latency FIFOs to complicated routing designs. Latency-insensitive channels can communicate arbitrary message types with all marshalling/demarshalling handled by the compiler. These latency-insensitive channels can be used for all kinds of communication in the target application, including FPGA on-chip communication, communication between an FPGA and a processor, and communication across FPGAs.

1The code and interface examples in this thesis use a simplified Bluespec syntax.
LEAP leverages latency-insensitive channels to abstract physical devices and platform services, such as across-platform communication and memory systems, enabling the service implementations to be changed, optimized, or replaced without modifying the user program. In addition, LEAP latency-insensitive channels enable communication between any two points in a design without explicitly connecting wires through the module hierarchy. This channel construction primitive greatly simplifies the implementation and optimization of operating system functionalities.

To facilitate the implementation of platform service libraries, LEAP additionally supplies a broadcast communication primitive: latency-insensitive rings. During compilation, ring stops are aggregated by name and connected in a unidirectional-ring topology via latency-insensitive channels. LEAP further supports client-server bidirectional communication for multi-client services using these latency-insensitive ring primitives, which provide a simple way to connect the clients of the target service with the server when the number of clients is unknown prior to compilation. Most LEAP services, such as the standard input/output (I/O) service, the statistics collection service, and the private memory service, are implemented with LEAP rings for client-server communication. In these services, each client of the target service is assigned with an unique ID. The clients send out requests, each tagged with the requester ID, to the server through one ring and the server broadcasts responses through another ring. Each client checks the requester ID tagged in every incoming response packet and then decides whether to take the response or forward the packet on the ring.

LEAP opts for ring-based topologies for service implementations because rings are lightweight, largely symmetric, reasonably fair, and easy to assemble, both at compile time and at runtime. However, as FPGAs and FPGA applications have scaled with Moore's law, the main flaw of ring-based topologies has been increasingly exposed: latency. In this thesis, we seek to alleviate the network latency issues present in scaled out FPGA memory systems through the construction of more complicated network topologies with a new latency-insensitive communication primitive, which will be discussed in Chapter 6.
2.1. The LEAP FPGA Operating System

2.1.2 LEAP Private Memory

To simplify the memory management tasks in FPGA programming, LEAP offers a general, in-fabric memory abstraction: the LEAP private memory [3]. LEAP private memories provide a simple latency-insensitive interface with three methods: read-request, read-response, and write, as shown in Figure 2-2. Programmers can instantiate as many LEAP memories as needed to store arbitrary amounts of data with arbitrary data types using a simple, declarative syntax, as described in the following example:

```verilog
// Instantiate two private memories to store $2^{28}$ and $2^{15}$ items.
// Each data item has 48 bits.
module mkModuleA();
    MEM_IFC#(Bit(28), Bit(48)) mem1 <- mkPrivateMem();
    MEM_IFC#(Bit(15), Bit(48)) mem2 <- mkPrivateMem();
endmodule
```

Each instantiated LEAP private memory represents a logically private address. LEAP memories also 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'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 interface 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 static random-access memory (SRAM), while a larger memory could be backed by a cache hierarchy and host virtual memory. In this thesis, we leverage the freedom of abstraction to build optimized memory hierarchies on behalf of the user.

At compile time, LEAP gathers memory primitives in the user program and instantiates 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 2-3 shows an example of a typical LEAP memory hierarchy which integrates four private memories instantiated in the user program. LEAP memory clients optionally receive a local cache, which is direct mapped and implemented using on-chip SRAMs. By default, the program-facing memory clients are connected with a centralized memory controller through a LEAP latency-insensitive ring network. The centralized memory controller offers a read/write interface to address spaces, serving as an abstraction layer of the low-level memory subsystem. 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 is uniquely tagged, enforcing a physical separation. The main memory of an attached host processor backs this synthesized cache hierarchy.
2.1. The LEAP FPGA Operating System

2.1.3 LEAP Compilation

A typical LEAP program is composed of latency-insensitive modules, which communicate with the rest of the system only by way of latency-insensitive channels. The goal of the LEAP compiler [31] is to map the latency-insensitive modules onto an environment, which may be a single FPGA or an aggregation of multiple connected FPGAs, and construct the physical implementation of the latency-insensitive channels connecting the mapped modules. During compilation, latency-insensitive modules are automatically connected through simple hardware FIFOs, if communicating modules are on the same platform, or through efficient, complex, inter-FPGA network, if communicating modules are on different FPGAs.

The LEAP compiler maps the target program onto a single or multiple FPGAs using the Bluespec compiler and FPGA-vendor tools as subroutines. The LEAP compilation flow is shown in Figure 2-4. First, the source code of the target program are passed to the Bluespec compiler. The LEAP compiler parses the intermediate files produced by the Bluespec compiler to construct a dataflow graph representation of the user program. Inside the graph, vertices correspond to latency-insensitive modules, while edges are the associated latency-insensitive channels. Then, the LEAP compiler maps the graph to the target execution environment consisting of single or multiple FPGA platforms based on a
platform mapping file, which specifies the associated platform for each latency-insensitive module. The platform mapping file can be either provided by the programmer or automatically generated by the LEAP compiler. After the latency-insensitive modules have been placed, the LEAP compiler constructs the physical network connecting these modules and produces a top-level Bluespec program for each FPGA platform. The generated programs are passed to the Bluespec compiler for a second time to generate Verilog code and then an FPGA-vendor tool chain to produce final physical implementations.

In this thesis, to optimize the network connecting LEAP memory clients on an application-specific manner, we extend this LEAP compilation flow by adding a series of compilation phases, which we refer to as the LEAP Memory Compiler (LMC). We will describe LMC in full in Chapter 7.

2.2 Extending LEAP Memories to HLS Applications

To simplify implementation tasks while having tight control over architectural design and optimizations, as mentioned in Section 2.1, LEAP uses Bluespec to assist its platform service implementations. Bluespec SystemVerilog (BSV) [12] is a high-level hardware description language (HDL) with syntax based on SystemVerilog and abstractions inspired by Haskell. Unlike C-based HLS programs, which are written in a sequential computational model and thus have difficulty expressing irregular and complex parallelism in memory hierarchies and on-chip networks, the behavior model of BSV programs is fundamentally parallel. BSV programs are implemented based on a set of guarded rules, which are called Guarded Atomic Actions and can be used to express complex concurrency in hardware [70, 71]. This rule-based programming paradigm has a steep learning curve but can generate efficient hardware with performance close to that of a hand-written, RTL-based design.

While LEAP services are written in BSV, they can be used by applications written in any synthesizable language, including RTL-based applications and kernels written in C/C++ and compiled through HLS. This means that designing FPGA programs can be further simplified while maintaining high performance by integrating HLS applications.
2.2. Extending LEAP Memories to HLS Applications

with a LEAP-generated platform. However, applications not written in BSV need to connect to LEAP services through shims, which may be a design burden for users without hardware experience. To improve LEAP usability for HLS programmers, we design wrappers that connect HLS kernels to LEAP memories through bus protocols and make some steps toward automating this integration process. Since HLS tools generally have limited support and optimizations for accessing external memories, integrating with optimized LEAP memories can provide a large performance benefit for HLS applications with high storage capacity demands.

The first set of HLS applications integrated into a LEAP-generated platform was done in collaboration with Felix Winterstein from Imperial College London and published in [95]. The HLS applications included in that paper as well as in this thesis were implemented by Felix and my contribution was to support the integration process. The HLS kernels are constructed using Xilinx Vivado HLS [94] (formerly AutoPilot [112] developed by AutoESL), which is one of state-of-the-art C-to-FPGA tools and has extensive support for our evaluation boards. In Vivado HLS, directives can be used to guide compiler optimizations and to specify the I/O protocols for the application. Our HLS applications use Vivado’s native bus protocol (the *ap_bus* protocol) to communicate with LEAP memories. The application kernels are first compiled to Verilog implementations through HLS. Then we use a Verilog wrapper to embed the generated kernels into the LEAP platform using BSV’s Verilog import feature. The wrapper acts as a bridge between HLS kernels and LEAP memories by converting bus-based transactions into latency-insensitive LEAP memory transactions.

To further improve the usability of LEAP memories, we also design wrappers to support HLS applications that use AMBA AXI4 (Advanced eXtensible Interface 4) protocol [6] to access external memories. In addition, we provide an automated flow to integrate applications that only uses a single LEAP private memory. Figure 2-5a shows an example of the automatically integrated system. The instantiated LEAP memory has multiple ports for applications with multiple kernels, each of which accesses the LEAP memory through a separate bus interface. The integration flow is shown in Figure 2-5b. First, the Verilog implementation of the target application core is generated through
Chapter 2. Background

Figure 2-5: Automating the integration of HLS applications and LEAP-generated platforms.

Vivado HLS. Then, the information of the application's memory buses is extracted from the core to construct the associated wrapper and the LEAP platform. The generated wrapper contains memory interface adaptors, which forward requests and responses between HLS kernels and the LEAP memory. Finally, the wrapped core is integrated into the LEAP platform and passed to the LEAP compilation flow discussed in Section 2.1.3 to generate an FPGA image. Future work will support applications that require multiple independent address spaces or coherent memories.

2.3 FPGA Platforms and Workloads for Evaluation

To evaluate the benefit of the proposed memory optimizations and abstractions, we will examine a diverse set of hand-assembled and HLS-compiled benchmarks with different memory access characteristics. Many of them are previously published FPGA implementations that were originally expressed in terms of generic request-response-based memory interfaces, and applying our optimizations requires no modification in the application source code.
2.3. FPGA Platforms and Workloads for Evaluation

2.3.1 Hand-Assembled Applications

The following applications are hand-assembled programs written in BSV:

Memperf: A kernel that measures the performance of a memory hierarchy by issuing memory requests with various data strides and working set sizes. It uses a single LEAP private memory. Memperf is bandwidth-intensive and can issue as many outstanding requests as the memory system permits.

Blocked Matrix-Matrix Multiplication (MMM): The MMM hardware [24] that we evaluate uses a block-style decomposition with its own internal block buffering. The two input and the output matrices are stored in separately initialized LEAP private memories. When calculating a single output block, the hardware pulls in a complete row and column of the source matrices, computing a streaming multiply-accumulate. The memory access pattern of MMM is fixed statically and does not vary with input.

Mergesort: Mergesort contains one or multiple highly parallel sorter engines, each of which sorts a memory array using the merge sort algorithm. Each sorter engine [33] 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 formed by multiple concatenated mergers. At runtime, the merger hardware repetitively picks a pair of lists and performs a single merge on them, propagating the result to the next merger. The item chosen at each list merge is data-dependent, causing the stored lists to drain at different and unpredictable rates. The hardware attempts to deal with this by observing the amount of data remaining in each list and fetching new data as the list drains. Thus, while the set of fetches produced by the merge sorter are determined statically, the order in which these fetches are issued by the hardware is data dependent. The merge operation and, therefore, the memory access behavior of mergesort is related to sparse matrix algebra [43], making this workload broadly representative of that class of algorithms.

Mergesort is bandwidth-intensive and can be scaled to consume almost any amount of bandwidth. Mergesort 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 sorter engines and banks, but if scaled to an extreme, results in large memory network queuing delays and degraded overall system performance. We will examine two configurations: a baseline configuration with a single LEAP private memory for each sorter and a configuration with two LEAP memories, which we refer to as banked.

H.264: H.264 is a video decoder that constructs new frames from portions of previously decoded frames stored in memory, a process known as inter-prediction. Pre-existing frames are generally fetched into the processing core in a raster-like order, left-to-right and top-to-bottom. However, there are significant sources of non-determinism in the fetch pattern. First, portions of the frame may not be inter-predicted, creating gaps in the memory access stream. Second, because there are several possible prediction modes, the memory access pattern may vary in stride at a fine grain. Our H.264 implementation [34] uses three separate LEAP private memories, one for the luminance field and two for the chrominance fields.

HAsim: HAsim [77] is a framework for constructing cycle-accurate simulators of multi-core processors. To model multiple cores, HAsim time-multiplexes components of a single processor, sharing these components among all modeled processors. Unlike processors, which are often sensitive to latency, HAsim's time multiplexed implementation makes it very latency tolerant. HAsim uses multiple LEAP private memories to model various structures of the processor, including caches, branch prediction tables, and translation lookaside buffers. The largest private memory used by HAsim captures the virtual memory state of the modeled processor.

Heat: A two-dimensional stencil code that models heat transfer across a surface. In stencil computations, each grid point is repeatedly updated with a function of its neighboring points in both time and space. Heat is embarrassingly parallel and can be easily divided among as many worker engines as can fit on the FPGA if a shared memory infrastructure is supported. 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
2.3. FPGA Platforms and Workloads for Evaluation

\[
\begin{align*}
\text{for } t = 0 \text{ to } T-1 \\
\text{for } y = 1 \text{ to } N \\
\text{for } x = 1 \text{ to } M \\
U[t+1,x,y] &= C_0 \cdot U[t,x,y] \\
&+ C_x \cdot (U[t,x-1,y] + U[t,x+1,y]) \\
&+ C_y \cdot (U[t,x,y-1] + U[t,x,y+1]) \\
\end{align*}
\]

end
end

end

Figure 2-6: A simple nested loop that models heat transfer from time 0 to time T on an \( M \times N \) grid. \( C_0, C_x, C_y \) are constants related to thermal diffusivity.

bandwidth-intensive with a strided access pattern, but with a high degree of locality. In the single-worker implementation, the worker accesses a LEAP private memory. In multi-worker implementations, each heat worker accesses a LEAP coherent memory.

Unlike above mentioned benchmarks, which are previously published implementations, heat is newly implemented to evaluate the performance of the proposed LEAP coherent memories. Figure 2-6 shows how the single-worker baseline performs a serialized 2-dimensional heat transfer computation. Since the pixel values at time \( t \) only depend on the values at time \( t-1 \), we allocate two frame buffers in memory: one to hold the current time step and another to hold the previous time step. To take advantage of locality and make the program cache-efficient, we interleave the two frame buffers and store them in row-major order. In addition, we decouple memory reads from pixel computation and memory updates, exploiting pipeline parallelism to overlap the memory access latency. To parallelize the heat transfer computation, the frame is divided into blocks. Each block is assigned to a worker engine, and the grid points at the borders of blocks are shared between workers. The data sharing is automatically handled by LEAP coherent memories. During an iteration, each worker sweeps through the assigned block, computes, and updates new values until it reaches a barrier point. Then it waits for the barrier signal, which indicates that all workers have completed the current time step, before it starts the iteration of the next time step.
2.3.2 HLS-Compiled Applications

The following applications contain computational kernels that are written in C++, compiled through Vivado HLS, and integrated with the LEAP platform through Vivado native buses:

**Prio:** An HLS kernel implementing a priority queue using a sorted, doubly-linked list. The application performs a sorted insertion for each new entry. *Prio* maintains a single queue and uses one LEAP private memory.

**Merger:** An HLS kernel that merges several linked lists together to form a sorted list. *Merger* [95] forms four linked lists in parallel from streams of random integers. After a constant number of inputs have been received, it repeatedly deletes the smallest head node among the lists until all lists are empty, producing a sorted output sequence. *Merger* uses four LEAP private memories.

**Filter:** An HLS kernel that implements a filtering algorithm [54] for high-performance K-means clustering, a widely used machine learning technique for unsupervised partitioning of a data set. K-means clustering partitions a multi-dimensional data set into K clusters such that the sum of the distance from each point to the centroid of the assigned cluster is minimized. The K clusters are represented by their geometrical centers (means). The task is to find a locally optimal placement for the K centers. A standard algorithm starts with a random placement of the centers, which is refined in several iterations until it converges to a locally optimal solution. During each iteration, the data points are assigned to the nearest center, forming K partitions, and then the mean of each partition is computed. These means form the next generation of centers.

A naïve implementation needs to compute $N \cdot K$ Euclidean distance calculations in each iteration, where $N$ is the number of data points. This has high computational complexity for large data sets. The filtering algorithm prunes the search space of the nearest centers by organizing the data points in a binary search tree (a 'kd-tree' [54]) and finding nearest centers using a tree traversal.
2.3. FPGA Platforms and Workloads for Evaluation

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 HLS core parallelizes the traversal loop and splits the tree into $P$ independently-processed sub-trees. Each partition traverses an independent sub-tree using a separate stack and maintains its own sets of candidates for the best cluster centers.

Our implementation is slightly different from the one proposed in [95], which uses LEAP coherent memories to manage the centroid information shared among partitions. In our implementation, the shared centroid information is handled inside the HLS core and filter only uses LEAP private memories. Each filter partition uses three LEAP private memories to store different data structures: a sub-tree, a stack, and candidate center sets.

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. The program visits every node of the tree. Similar to filter, the HLS core consists of $P$ parallel partitions and sub-trees. Each partition manages the tree traversal with a stack, which is implemented with a pointer-linked list. Each list node contains a pointer to the sub-tree. 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 stack and the sub-tree of each partition are stored in a separate LEAP private memory.

The HLS-compiled workloads mentioned above chase data-dependent pointers and thus have limited ability to produce multiple, parallel memory requests. As a result, the performance of these HLS workloads is more sensitive to memory latency compared to other workloads we study.
2.3.3 FPGA Platforms

In this thesis, we run the workloads on a set of FPGA platforms. Since the benchmarks we study are written on top of LEAP, we can port implementations among different FPGA platforms without source modification by simply re-targeting the LEAP compilation flow to a new platform. Re-targeting a program does not change the functional behavior of a program, just as recompiling a C code for a new machine does not change the functional behavior of the C code.

Table 2-1 lists the FPGA platforms we use for evaluation. These platforms have different amounts of FPGA resources and memory hierarchies, which can influence the performance of FPGA applications as well as our optimization decisions.

Nallatech ACP: The Nallatech ACP platform has a pair of Virtex-5 LX330T FPGAs [100] that plug directly into an Intel Front Side Bus (FSB) socket as compute modules. The two FPGAs on the ACP are connected via a high-speed LVDS bi-directional interconnect. Each FPGA is attached to 8 MB of SRAM.

Xilinx ML605 [102]: The Xilinx ML605 is a Virtex-6 based FPGA board consisting of an LX240T FPGA [101] and a single DDR3 controller connected with 512 MB of DRAM.

Xilinx VC707 [103]: The VC707 is a widely deployed evaluation board built with a Virtex-7 VX485T FPGA [97]. The VC707 contains a single DDR3 controller connected with 1 GB of DRAM.
2.3. FPGA Platforms and Workloads for Evaluation

**Xilinx VC709 [104]:** The VC709 platform includes a Virtex-7 VX690T FPGA [97] and two DDR3 DRAM controllers, each connected to 4 GB of DRAM.

We connect the above Xilinx evaluation boards to a host processor through a PCIe interface. 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. The networked FPGAs communicate using two bidirectional 10 Gbps SERDES links managed by LEAP.
This thesis aims to ease FPGA programming while producing high-performance solutions by examining resource-aware and application-aware optimizations that can be performed under memory abstractions and introducing a build procedure to automate these optimizations. In this chapter, we summarize prior work that are conceptually related to this thesis. Section 3.1 and 3.2 introduce existing FPGA high-level programming environments and HLS tools that try to simplify FPGA application development, and the rest of the chapter focuses on FPGA memory systems, including memory abstractions, coherency management, and optimizations. Optimizations in the fields of processor architectures and embedded SoCs we take inspiration from will also be discussed.

### 3.1 FPGA High-Level Programming Environments

Due to the lack of high-level application development environments, FPGA programmers have traditionally spent a lot of design effort managing an entire stack from application kernels to device controllers and reengineering the stack when porting the program to a different FPGA. In order to reduce the burden of programming FPGAs by improving FPGA usability and application portability, in addition to the LEAP framework introduced in Section 2.1, several other FPGA operating systems and middleware services have been proposed.

BORPH [85] treats FPGA programs as normal processes is a standard Linux system and extends Linux services, including standard I/O and file system support, to these hard-
ware processes running on FPGAs. Under BORPH, hardware processes on the FPGA and software processes on the host processor are treated equally and can communicate with each other through file pipes. Hthreads [7] offers an abstract, multithreaded programming model that includes operating system support and middleware service abstractions. The Hthreads compiler and libraries allow programmers to create threads that can be compiled to run either on a central processing unit (CPU) or on an FPGA and extend standard thread communication and synchronization services across the CPU/FPGA boundary. Similar to LEAP, BORPH and Hthreads provide portable service abstractions that facilitate the development of applications involving software/hardware co-design for hybrid CPU/FPGA architectures. However, unlike LEAP, these programming environments only provide basic memory abstraction and memory management schemes to utilize FPGA physical memories while focusing more on adding communication support.

The RC Middleware (RCMW) [57] is another example that improves the portability and productivity of FPGA applications through middleware abstractions. RCMW abstracts away vendor-specific platform details, including memories and interconnects, and presents an application-centric view of available resources. It simplifies application development by handling the mapping of configurable user interfaces to physical platform resources, which includes multiplexing and arbitration when physical resources are shared among multiple user interfaces. Although RCMW offers portable platform abstractions, it lacks application-specific platform optimizations like what LEAP provides to further improve program performance.

### 3.2 FPGA High-Level Synthesis Tools

In addition to the application programming environments described in Section 3.1, there is another class of work attempting to simplify FPGA programming by providing tools to automatically map software source code to FPGA implementations. For example, several C-to-FPGA HLS tools, such as Xilinx Vivado HLS [94] discussed in Section 2.2, the Riverside Optimizing Compiler for Configurable Computing (ROCCC) [93], and LegUp [13], have been developed to generate FPGA designs from C-based programs.
3.2. FPGA High-Level Synthesis Tools

Vivado HLS, ROCCC, and LegUp are all built with an open-source LLVM (Low Level Virtual Machine) compiler infrastructure [87], which enables software optimizations such as dead-code elimination and loop-unrolling. Although these HLS tools can extract some parallelism from C programs and convert them to hardware implementations, producing high-performance designs usually relies on programmers to manually specify more aggressive optimizations through program annotation, which may be difficult for programmers with little hardware design knowledge. One way to remedy this limitation is to use domain-specific frameworks [40, 64, 80], which only support a particular type of applications, such as machine learning or image processing algorithms. By leveraging domain-specific properties, such as computational parallelism or memory access patterns, more sophisticated optimizations can be automatically performed to generate high-performance hardware.

Unlike previously described programming environments, many of these HLS tools focus more on kernel generation instead of full system design. As mentioned in Section 2.2, we extend LEAP memory services to application kernels compiled using Vivado HLS [94] integrating the kernels with the LEAP platform through standard bus protocols. The kernels generated by other HLS tools can also take advantage of optimized memory services provided in LEAP by applying a similar integration process as long as these kernels can communicate through standard buses.

OpenCL (Open Computing Language) [88] and OpenCL-to-FPGA compilation frameworks [23, 48, 82] take a slightly different approach to aid the development of FPGA applications by providing a programming model and HLS tools with better support for system integration. OpenCL is an open standard for parallel computing on heterogeneous systems consisting of CPUs and a variety of accelerator platforms, such as graphics processing units (GPUs), digital signal processors (DSPs), and FPGAs. OpenCL defines a programming model that comprises a host program running on a general-purpose CPU for setting up computation and a set of kernels executing on one or multiple hardware accelerators. OpenCL uses a C-based programming language with extensions for parallelism and defines a set of application programming interfaces (APIs) to control the communication between the host and accelerator kernels as well as the
Chapter 3. Related Work

communication among the kernels. Both Xilinx and Altera (now part of Intel) have produced OpenCL-to-FPGA tool flows [23, 48, 82] that automatically convert OpenCL programs to implementations targeting CPU/FPGA platforms. For example, the Intel FPGA SDK (software development kit) for OpenCL [48] provides an offline compiler, which is also based on the LLVM compiler framework, to transform OpenCL kernels to pipelined, optimized hardware circuit. The host program is compiled using a standard C compiler and then linked with compiled kernels. We believe OpenCL is suitable for kernel-based applications, while LEAP (with our extensions included) is able to target a more general class of programs.

3.3 FPGA Memory Abstractions

Traditionally, the construction of FPGA memory hierarchies has usually been left as a painful exercise to the programmer. In response to the increasing complexity of memory systems, FPGA vendors have begun to provide memory controllers, for example Xilinx Memory Interface Generator (MIG) [99]. Programmers instantiate a physical memory controller, and tie their program logic directly to the instantiated controller, simplifying the overall design experience. Although memory controllers are upgraded by the vendors over time and maintain nearly constant interfaces across memory and FPGA generations, these memory controllers provide just a basic implementation – advanced features like caches are still left to programmers.

Recent research into FPGA programming and architecture has suggested that programs can benefit, in some cases, dramatically, from improved memory system support. To shorten development time, several FPGA memory abstractions have been proposed to virtualize external memories, enabling portable and reusable memory management and optimizations across platforms and applications. The LEAP private memory discussed in Section 2.1.2 is one example. RCMW [57] provides two kinds of abstract memory interfaces – the burst and FIFO interfaces – and a handshaking protocol to arbitrate memory accesses. Each interface controller comes with a configurable transfer buffer for memory messages. RCMW handles the mapping of these memory interfaces to physical
3.3. FPGA Memory Abstractions

memories for programmers to improve productivity but it lacks memory optimizations such as on-chip caching or customized memory networks.

The CoRAM [19] architecture provides an abstract memory interface similar to that of the LEAP private memory to decouple kernel-based computation from memory management handled by control threads. Control threads are programmed using a C-like language to describe fetches to the external memory. These fetches are executed in a streaming fashion and stored in on-chip SRAMs for access by program logic. The control flow between the fabric and control threads permits a degree of program-dependent dynamic behavior in the fetch stream.

The LEAP memory with our proposed prefetcher\(^1\) inserted is similar to CoRAM in both on-chip caching and prefetching. The chief weakness of CoRAM lies in the difficulty of describing the memory access pattern. CoRAM works very well for deterministic access patterns, like matrix-matrix multiplication. However, for data-dependent or complex access patterns, a C-like fetch description language may be inadequate. For example, consider the strongly data-dependent access pattern of an H.264 decoder: memory access generation is tightly coupled to states and logic in the FPGA fabric, rendering it difficult to describe the fetch pattern in an external language. In addition, CoRAM provides no support for shared memory within or among FPGAs.

In this thesis, we adopt the LEAP private memory as our basic memory abstraction, performing various optimizations underneath and backing it with a customized memory network. Similar to the LEAP memory, CoRAM does not define the memory hierarchy or network backing its programmer interface, and therefore could make use of our optimization approach. There is an ongoing work that integrate CoRAM with LEAP private memories by taking CoRAM control threads as user-specified prefetchers to preload data into LEAP caches.

\(^1\)The prefetcher design will be described in detail in Section 5.1.
3.4 Managing Coherency and Synchronization on FPGA

Distributed memory management and synchronization on FPGAs have been examined in several previous projects. Unlike our proposed shared memory abstraction, which is intended to support general FPGA programming, all previous works of which we are aware target coherence and synchronization among soft cores [59, 66] implemented on the FPGA. Most implementations of coherent soft processors make use of snoopy protocols, though a directory-based protocol has been examined [65]. In contrast to our scalable, network-based implementation, all of these implementations appear to rely on crossbars to communicate between caches in relatively small, single coherence domains. These structures are neither scalable nor can they be easily partitioned among multiple FPGAs. Moreover, since existing coherence work targets multi-core processors, existing coherence implementations assume fixed coherence domains among the processors, rather than granting the programmer freedom to describe free-form coherence domains. Although we have not yet attempted to use our work in the context of soft processors, we believe that our coherent cache interface could be used to support a distributed soft multi-core architecture using the testAndSet atomic operation.

Like coherence, most efforts at implementing synchronization in FPGAs have occurred in the context of soft-cores, either to provide synchronization mechanisms among the cores [63, 67] or between the cores and hardware accelerators [7]. Some of these implementations are bus-based [7, 98], limiting scalability within an FPGA and preventing expansion to multiple FPGAs. Since the existing implementations are processor-specialized, their interfaces typically involve memory addresses which are used to denote mutexes. As a result, these implementations may require more area to implement than our logic-integrated locks (as much as 3× in the case of [67].) Existing implementations provide only lock/mutex management, mostly for the implementation of soft-processor atomic operations. Barriers must be implemented in software on top of mutexes, which may limit performance.

These traditional FPGA-based processor infrastructures [59, 63, 66], are also sufficiently abstract to be compatible with our optimizations, especially the cache network
3.5 Memory Optimizations

optimizations. These multicore systems could 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 is typically symmetric when viewed across many workloads. Thus, the benefit of application-specific optimizations for such symmetric systems is less obvious.

A second set of work focuses on building a cache coherent interface between an accelerator FPGA and host processors. The most recent of these works is Intel QuickAssist QPI-based FPGA Accelerator Platform [72]. QuickAssist manages both a coherence interface to an attached processor and the FPGA-local DRAM in-fabric, presenting a single processor-coherent memory interface to the FPGA. Unlike our proposed coherent memory interface, which is designed to allow multiple distributed memory interfaces, QuickAssist supports only a single coherence interface and does not admit of further scaling, due to the difficulty of meeting coherence-protocol-level timing in the FPGA. If these timing issues could be resolved, our coherence interface could be used to bridge the QuickAssist infrastructure and multiple FPGAs, providing processor-FPGA coherence across a network of FPGAs.

3.5 Memory Optimizations

Memory is fundamental to the performance of many 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 fixed at design time and determined through human implementation effort due to the high production volume of these systems and their symmetry. On the other hand, in lower volume architectures, like embedded SoCs, which have asymmetric use cases, design automation is often employed to optimize the memory system topology, producing high-performance solutions while meeting tight time-to-market constraints. With FPGA memory abstractions, many of the optimization and automation concepts used in both kinds of the systems can be applied to efficiently customize the FPGA memory system for a given application.
3.5.1 Memory Optimizations in Processors

The similarity between LEAP's memory system and that of general-purpose processors enables us to augment the memory hierarchy with optimization techniques resembling those used in processors.

Prefetching is one such technique. In processors, hardware prefetching mitigates the impact of cache misses by predicting program behavior and issuing memory reads in advance of program execution. Prefetching schemes attempt to predict two properties of data: \textit{which} data will be used and \textit{when} it will be used. Consequently, prefetching schemes are judged based on accuracy, fetching useful data, and timeliness, fetching data into the cache prior, but temporally close, to its use.

The simplest prefetching technique is next sequential prefetch, which issues a prefetch request to the next cache line \((L + 1)\) when the current cache line \((L)\) is accessed [50, 84]. Prefetch-on-miss issues a prefetch request on every cache miss, while tagged-prefetch issues requests both on a cache miss and on the first access to a prefetched cache line, a prefetch hit, to further reduce the number of misses in a sequential access stream [91]. To enable tagged-prefetching, a tag bit is added to each cache line, marking prefetched cache lines that have not been accessed.

To enhance accuracy, hardware prefetching methods dynamically attempt to learn access patterns from program access streams. However, the learning process is complicated by the interleaving of streaming memory accesses with other streaming accesses and non-predictable data fetch. To separate different streaming accesses and to filter out the non-predictable accesses, extra information, such as the program counter (PC) of load instructions [29, 37] or the memory region of the target address [45, 74], is used to disambiguate the program memory access stream. Once memory access streams are separated, a prefetch learning algorithm [37, 45, 49, 53] can be applied to each stream independently. Stride-prefetching is the most common of these algorithms. When the prefetcher learns the stride pattern, it issues memory accesses for \(L + s, L + 2 \cdot s, \ldots, L + d \cdot s\), where \(L\) is the current cache line, \(s\) is the detected stride, and \(d\) is the prefetch
3.5. Memory Optimizations

degree, the number of issued prefetch requests. The $d$ parameter is also called look-ahead distance [16], and may be adjusted to improve timeliness.

Recent prefetching schemes, such as Markov prefetching [49] and delta correlation prefetching [53], focus on detecting more complex memory patterns. To reduce the area complexity of these prefetchers, a Global History Buffer (GHB) [69] uses shared memory structures and linked-lists to store long access histories. Díaz et al. [27] extend the GHB by linking different local streams together to further increase accuracy and timeliness.

In addition to prefetching, large on-chip cache architectures also have been extensively studied in the field of processor architectures [4, 56]. Kim et al. [56] explore the design space for wire-delay dominated on-chip caches and propose large on-chip cache architectures with multiple banks and non-uniform access latencies. Agarwal et al. [4] reduce cache access delay to meet target frequency by pipelining and aggressively banking the cache. In this thesis, we use similar pipelining and banking techniques on top of BRAM primitives to construct large on-chip storage. Memory banking techniques have been used in existing FPGA-based architectures for higher bandwidth instead of reducing wire delays. In addition, we leverage the freedom of constructing FPGA-based caches tailored to the target application and study the application-specific cache design tradeoffs, which have not been well explored in the processor related research. For example, an application might benefit more from a small cache with shorter access latency than from a large banked cache with longer access latency, or vice versa. In Section 5.2, we study cache design tradeoffs among access latency, operating frequency, and cache capacity for several FPGA applications with different memory access behavior.

In Section 5.2, we also explore indexing algorithms for caches with non-power-of-two sizes. Existing non-power-of-two indexing algorithms for processor memories, such as the prime number indexing schemes [39, 60] and the Arbitrary Modulus Indexing [26], are proposed to reduce the number of address mapping conflicts. Software memory access strides usually have power-of-two divisors; therefore, using a non-power-of-two modulus (especially a prime modulus) for indexing effectively reduces the number of common divisors of the index modulus and memory strides and thus reduces the number of conflicts. In the thesis, we construct caches with non-power-of-two sizes for the purpose
Chapter 3. Related Work

of maximizing BRAM utilization instead of minimizing cache conflict misses. We target cache sizes that are products of a small integer and a power-of-two number and provide an indexing scheme that can be implemented with a lookup table and bitwise operations, eliminating the needs to use any adders, multipliers, or dividers.

Our technique of memory address interleaving across multiple memory controllers also 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 [46]. An example of memory interleaving is the SGI Challenge [38] 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 coherent cache network optimization introduced in Section 4.2 resembles some of the coherence architectures developed as part of the Hector project [30]. 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 and construct low-latency networks asymmetrically, if such asymmetry benefits a particular program.
3.5. Memory Optimizations

3.5.2 Memory Network Optimizations in Embedded SoCs

In the embedded domain, multiple accelerator devices are used to meet performance and energy targets. Multiple automated methods [21, 51, 52, 111] 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. A crossbar consists of multiple buses operating in parallel, providing better performance with higher area and power costs compared to a shared bus. As with our work in automatic synthesis of low-latency memory networks, in [51, 52], the mixed integer linear programming (MILP) formulation has been employed to optimize the network topologies, given some performance characteristics and goals of the accelerators in the target system and other design constraints such as area and operating frequency.

The prior work on SoC-style network synthesis primarily targets multimedia applications, which usually can be characterized by regular memory access patterns during compile-time analysis [21]. These works typically assume that the latency and bandwidth requirements from accelerator engines, either obtained from static analysis or provided by application designers, are given as as input constraints to the network construction algorithms, which aims to find an optimal network minimizing area/power cost or maximizing frequency. In this thesis, in order to target more general and more complicated applications, we propose a compilation flow that includes FPGA-based runtime instrumentation to characterize the latency and bandwidth demands from each processing engine. The instrumentation results are used as feedback to guide our network synthesis algorithms. In addition, as we will show in Section 7.3, the area cost of our memory network is small compared to the cost of caches and memory controllers. Therefore, instead of minimizing latency cost, our network synthesis focuses on minimizing the network latency impact on program performance.

Another major difference between our work and prior network synthesis studies that focus on constructing SoC-style networks lies in the choice of network topologies. SoC-style networks are intended to support memory accesses by accelerator engines and
memory-mapped communications between accelerators ganged together to perform some task. This requirement results in 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 (such as tree-based networks constructed with simplified N-to-1 crossbars), in turn improving key metrics such as area, frequency, and energy.

3.5.3 Memory Optimizations on FPGAs

Much recent work on FPGA-based memory optimizations has gone into the microarchitecture of shared caches on FPGAs [18, 25] and the construction of multiple-level memory hierarchies [18, 62], all of which build user-level memory interfaces and use off-chip storage for the second-level memory. None of these works explore the design space of second-level on-chip caches, even though this is a common technique in processor architecture. We are also not aware of any work that explores the microarchitectural tradeoffs of large on-chip caches on the FPGA. In addition, these works generally assume a fixed, symmetrical, program-invariant memory topology, while a major part of this thesis focuses on optimizing the memory topology on a per-application basis.

Dessouky et al. [25] propose a dynamic on-chip memory management unit (DOMMU) as an alternative means for constructing large storage elements on the FPGA. The work facilitates the dynamic sharing of on-chip memory among several processing elements (PEs), according to their dynamic memory requirements. In DOMMU, PEs interface to the memory subsystem by requesting and subsequently freeing memory regions. Like LEAP private memories, client regions are disjoint among the PEs. A key difference between our scalable cache primitives, which are discussed in detail in Section 5.2, and DOMMU is that DOMMU maintains the single-cycle BRAM interface. Single-cycle response latency requires that DOMMU implement an expensive internal crossbar between the PEs and the BRAM store, fundamentally limiting the scalability of DOMMU in
3.5. Memory Optimizations

terms of both the number of PEs and BRAMs it can support and operating frequency. We can support at least an order of magnitude larger memory systems, at the cost of a slightly different program-facing interface and a slightly longer access latency.

Choi et al. [18] incorporate the multiporting approach proposed by LaForest et al. [58] into a cache hierarchy comprised of a shared, multiported first-level cache backed by an off-chip RAM. Although the multiporting technique proposed by LaForest et al. is excellent for implementing small and low-latency storage structures, it exhibits limited scalability in terms of cache capacity and the number of clients due to the complexity of managing operation ordering among the ports. The high-level architecture of LEAP private memories is similar to the one in [18], but we adopt a more scalable microarchitecture for LEAP caches. Matthews et al. [62] also explores first-level cache microarchitecture, focusing on set associativity and replacement policy in addition to multiporting. Like the exploration of cache design tradeoffs discussed in Section 5.2.6, Matthews et al. find that increasing cache complexity, especially with respect to cache sizing tends to decrease operating frequency, although the relative decline reported in [62] is smaller than the decline reported in Section 5.2.6.

Beyond these architectural efforts, researchers have also explored the implementation of SoC-style memory networks in the context of FPGAs [17, 22], 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 within the application to achieve high performance.

The Shrinkwrap compiler [20], which is layered upon the CoRAM memory abstraction [19], automatically constructs application-specific memory networks based on the connectivity information, memory traffic direction, and bandwidth demands preserved by CoRAM control threads. Similar to our automatic network synthesis procedure, Shrinkwrap also constructs parameterizable, asymmetric tree-based network to efficiently handle the traffic between on-chip storage and off-chip DRAMs. Shrinkwrap optimizes memory networks by exploring throughput-area tradeoffs through thread aggregation
(or disaggregation), which combines (or decomposes) the assignment of control threads (and the associated CoRAMs) to on-chip clusters. Similar to our memory interleaving technique, thread disaggregation improves the overall throughput by adding extra end-points into memory interconnects. However, unlike our work, Shrinkwrap does not take network latency into consideration when optimizing memory networks, while focusing on area reduction through decoupled read-only and write-only clusters and interconnects. In addition, in Shrinkwrap, the memory traffic information of the target application is obtained statically analyzing the C-based software control threads, while we rely on FPGA-based runtime program instrumentation.
Parallel programming is widely used in many fields of science and engineering, including fluid dynamics, geometric modeling, and image processing, to quickly solve large-scale problems. This popularity is due, in part, to strong system-level support and powerful abstractions for parallel programming provided by general-purpose architectures. Threading libraries enable lightweight partitioning of programs, and the shared-memory programming model provides these threads a single view of memory. In addition, synchronization libraries help software programmers coordinate computation among the threads.

Abundant parallelism and fast communication make FPGAs attractive for accelerating parallel programs. In addition, recent advances in FPGA compilation techniques allow programmers to easily partition designs across FPGAs [32], enabling the use of multiple FPGAs as a computation platform, which is an ideal target for parallel programming.

Although FPGAs intrinsically support parallel program descriptions, mapping parallel algorithms to FPGAs remains difficult, due to the lack of good libraries for shared memory and synchronization. As a result, FPGA programmers are exposed to the complexity of distributed coordination and have to explicitly handle data sharing among processing engines [44, 83]. This requires significant design effort and thus prolongs the development time of FPGA-based parallel algorithms.

In this thesis, we seek to simplify FPGA-based parallel programming by maintaining the coherency and consistency of accesses to shared memory resources on behalf of programmers. The ultimate goal of this thesis is to automatically construct a memory
hierarchy optimized for the target application. To achieve this goal, the first step is to provide memory abstractions enabling application-specific optimizations. Since the LEAP private memory abstraction only supports accesses to private memory spaces, to facilitate the development of shared memory applications, we provide a shared memory abstraction with coherency and consistency support.

We propose the LEAP coherent memory, which manages memory coherency while presenting users with a simple interface similar to that of FPGA BRAM blocks and LEAP private memories. LEAP coherent memories provide the illusion of unlimited virtual storage, while automatically managing multiple coherent caches and multiple coherence domains across multiple FPGAs. Coherency comes with a need for consistency. To improve execution efficiency, our coherent memory implementation allows independent memory requests to be served out of order and maintains a weak consistency model. Since programmers sometimes need to enforce the ordering of certain memory operations, the interface of LEAP coherent memories provides non-blocking memory fences that enable the support of various consistency models in the FPGA. The implementation details of LEAP coherent memories are included in Section 4.1.

Our coherence scheme automatically scales across FPGAs because it makes use of latency-insensitive channels and the LEAP multi-FPGA compiler described in Section 2.1.3 can be leveraged to partition designs across different configurations of FPGAs. However, when spanning multiple FPGAs, the baseline implementation described in Section 4.1 offers limited performance benefit due to long inter-FPGA latency and the inability to utilize board-level memory resources on remote FPGAs. To tackle these performance challenges, we apply memory interleaving techniques to partition the shared memory address space into multiple disjoint regions. We also adopt a hierarchical-ring topology in the partitioned coherent cache network to reduce inter-FPGA traversals. The proposed coherent cache network optimization is described in Section 4.2.

Distributed coordination is fundamental in shared-memory programs. In general-purpose processors, locks and barriers are often handled through memory, since memory is usually the only mechanism available for communication. In FPGAs, other direct communication mechanisms are available. Thus, it is more efficient on FPGAs to provide
4.1 LEAP Coherent Memory

lock and barrier services outside of shared memory. Section 4.3 describes the design of our lock and barrier services.

We evaluate the performance of our coherency and synchronization primitives using multiple benchmarks which we map to both single and multiple FPGAs and discuss the results in Section 4.4. Our proposed primitives enable the concise description of parallel algorithms: these benchmarks require around 350 lines of codes and were written in only a few hours, which we argue is a substantial reduction in code complexity and development time. For the heat workload introduced in Section 2.3.1, LEAP coherent memories with coherent caches provide up to $3.8 \times$ speedup over the single-worker baseline and run $2.6 \times$ faster than a centralized-cache implementation on a single FPGA. When spanning two FPGAs, coherent memories take advantage of increased DRAM bandwidth and provide up to $7.6 \times$ performance improvement against the single-worker baseline. The proposed synchronization service outperforms the traditional through-memory primitives by $1.8 \times$ in the shared-queue benchmark, and our barrier primitive also achieves $340 \times$ higher throughput than that of an existing FPGA barrier primitive.

4.1 LEAP Coherent Memory

LEAP private memories are a powerful abstraction for describing private, independent memory spaces. However, they are insufficient as a shared memory programming substrate because programmers are responsible for managing all data sharing. For example, a user may instantiate a single large programmer-multiplexed private memory that connects to multiple processing engines. This kind of implementation is undesirable because it scales poorly: a single memory forces the serialization of requests. Moreover, if a processing engine is on a remote FPGA, the inter-FPGA latency, which is at least an order of magnitude longer than the intra-FPGA latency, increases the memory access latency and may have great impact on performance. In this work, we solve these performance issues by introducing a distributed, coherent cache local to each accessor. We extend the baseline LEAP private memory to manage a coherent cache and refer it as a LEAP coherent memory.
interface COH_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);
    // t_REQ r := {READ, WRITE, FULL}
    method void fence(t_REQ r);
    method Bool requestPending(t_REQ r);
endinterface

Figure 4-1: The LEAP coherent memory interface extended from the private memory interface with extensions highlighted.

LEAP coherent memories retain all the properties of LEAP private memories: arbitrary amounts of data with arbitrary types, virtualization, a simple user interface, and caching. Like the shared-memory abstraction in general-purpose machines, LEAP coherent memories are largely transparent to programmers, enabling programmers to focus on designing parallel algorithms.

4.1.1 Coherent Memory Interface

Figure 4-1 shows the interface of LEAP coherent memories. This interface is an upward compatible extension of the baseline private memory interface with memory fence requests added to support various consistency models. As in the processor’s shared memory, fences are necessary because our coherent memory implementation maintains a weak consistency model that allows serving memory requests out-of-order to improve performance. Usually this out-of-orderness is okay, but sometimes programmers need to enforce ordering for correctness. We provide three kinds of fences: write, read, and full fences to support various consistency models, allowing programmers to manage program ordering. A full fence ensures that all read and write requests prior to the fence will be processed before any request that comes after the fence is processed. Similarly, a write fence enforces the ordering of write requests issued before and after the fence request, and a read fence enforces the ordering of read requests. All memory requests, including reads, writes, and fences, are pipelined, and users are allowed to issue read or write requests after fence requests to maintain pipeline parallelism. Since supporting fences adds additional logic and latency to the cache access pipeline, we also provide a
4.1. LEAP Coherent Memory

second consistency mechanism: a requestPending signal indicating whether there is an incomplete request. This can be seen as a cheaper, but coarser consistency operation, which is useful for synchronization among multiple coherent memory clients. For clients that only require coarse consistency guarantees, this consistency mechanism is enabled to simplify the cache implementation.

While general-purpose processors usually assume a single coherence domain in which all threads share a single global memory, we take advantage of FPGA flexibility and allow FPGA users to declare multiple coherence domains with independent memory address spaces using compile-time specified domain names. Disjoint coherence domains do not interact. The following example shows how to specify a coherence domain and instantiate associated coherent memories:

```cpp
// Instantiate a coherence controller and three coherent memory clients
// Domain name: "IMG_A", address size: 22-bit, data size: 8-bit
mkCoherenceController("IMG_A", 22, 8);
COH_MEM_IFC#(Bit(22),Bit(8)) client1 <- mkCoherentMem("IMG_A");
COH_MEM_IFC#(Bit(22),Bit(8)) client2 <- mkCoherentMem("IMG_A");
COH_MEM_IFC#(Bit(22),Bit(8)) client3 <- mkCoherentMem("IMG_A");
```

4.1.2 Coherence Protocol

LEAP coherent memories specify an abstract shared memory interface and the underlying memory system can be implemented in different ways. The coherent memories currently implement a snoopy protocol. Our coherent memories support multiple coherence domains, separating FPGA programs that touch different address spaces. In addition, since FPGA memory systems can be targeted to each program individually, programs that do not require coherence can simply use private memories. This limits the number of clients of a memory resource to the exact number of those who actually need to share data and consequently limits the performance impact of the snoopy protocol.

Coherent memories in the same coherence domain are connected via ring networks and use a global ordering point to ensure that all coherent memories see the coherence requests in the same order [5, 10, 61]. Coherence requests are sent to the global ordering point where they are activated. The ordering point then broadcasts these activated
requests on the ring. Each coherent memory client and the backing, next-level memory snoop the activated requests, taking actions depending on the local coherence state of the target address.

We implement a MOSI protocol, which is based on the protocol specified in gem5 [11], to reduce the frequency of cache write-backs. MOSI stands for modified, owned, shared, and invalid, the four possible steady states of a cache line. A cache line is in the M (modified) state if it has a valid dirty copy of the data block and no other caches have a valid data copy. A cache line is in the O (owned) state if the cache is one of the several with a valid dirty copy of the data block and it is the only one that owns the data. If the cache line is in the M or O state, then it is the owner of the data block and is responsible for writing back dirty data to memory if the line gets evicted. In addition, the protocol supports data forwarding, which means the owner of a data block is responsible for responding to other caches’ snoop requests targeting the same block. A cache line is in the S (shared) state if it is one of the several containing a valid copy of the data block, which may be either dirty or clean. A cache line is in the I (invalid) state if the cache does not have a valid copy of the line. To perform a write operation, the cache line must be in the M state, while read can be performed if the cache line is in the M, O, or S state. The next-level memory keeps a directory that stores the coherence ownership information for each memory address, indicating whether the data is owned by one of the coherent caches or not. The directory, combined with the cache ownership information, ensures that only the actual owner of the data responds to a snoop request, without requiring fine-grained coordination among the controller and caches.

To speed up read-modify-write and to simplify the controller implementation, we modify the protocol in a way that a cache line in the I state is automatically upgraded to the M state on a read request to the next-level memory when none of other caches has a valid copy. In this way, the subsequent write can be performed directly without notifying other caches. In addition, since forwarding data from coherent caches is faster than from the next-level memory, we further reduce the frequency of data forwarding from memory by supporting automatic state transition from the I state to the O state on a read to the memory when other caches do not own the data block but may have
4.1. LEAP Coherent Memory

Figure 4-2: Coherent memory architecture.

a valid copy. However, this means the new M state and the O state (that is transited from the new M state or automatically upgraded from the I state) may contain clean data. To eliminate unnecessary data write-backs, each cache line has an additional dirty bit to support clean write-back, in which only the ownership information is written back to the controller. Our M state can be seen as a combination of the M state and the E (exclusive) state in the standard MOESI protocol.

4.1.3 Coherent Memory Architecture

Figure 4-2 shows the coherent memory architecture, which is integrated into the LEAP private memory hierarchy. In this example, there is one coherence domain that contains three coherent memory clients and one coherence controller. A client accessing a private memory is also included in the figure for comparison. The coherence controller serves as a global ordering point as well as the interface to the next-level memory. To prevent deadlocks, coherent memory clients and the controller within the same coherence domain
are connected via three rings: the unactivated request ring, the activated request ring, and the response ring. As mentioned in Section 4.1.2, the next-level memory has to store both data and the coherence ownership information for each memory address. To simplify the design, these stores are themselves implemented as LEAP private memories. This gives coherent memories high scalability in terms of address space size because we can leverage the existing memory hierarchy and its virtual memory support. In addition, since the coherence ownership access usually has high locality, utilizing the private memory cache hierarchy makes the coherent memory system more efficient.

4.1.4 Coherent Memory Client Microarchitecture

As shown in Figure 4-2, a coherent memory client contains a marshaller, a coherent cache, and a router. The cache line size itself is adjustable as a parameter, allowing our coherent memories to support any data size and line configuration. To ensure atomicity, the data size of client requests is constrained to be smaller than or equal to that of a cache line. If the request size is smaller than the line size, a marshaller is instantiated to handle partial reads and writes using masks.

Figure 4-3 shows the microarchitecture of the direct-mapped coherent cache and the router. The cache is designed to serve multiple local requests from the client and network requests from remote caches. A completion table is added in the router to store the metadata of network requests that need to be snooped. The size of the completion table controls the number of snoop requests allowed to enter the cache pipeline. Each component in the cache is pipelined to achieve high throughput, and we allow requests to different memory addresses to be served out-of-order to exploit operation-level parallelism.

To improve throughput, coherence network transactions are split-phase. Decoupling requests and responses increases the complexity of handling cache misses, since there might be multiple outstanding requests targeting the same address. Transient states must be handled during the transition from one steady state to another. For example, a cache line needs to move from the I state to the M state on a write miss before the write operation can be performed. To obtain the exclusive ownership of the data block, the cache first sends out a write miss request to the unactivated request ring and changes the
state of the cache line to the transient state IM_AD, which means the cache line is waiting for the data response and its own activated request before it can be transited to the M state. If its own activated request arrives first, the cache line is transited to the IM_D state, waiting for the data response. If the cache then sees another activated read miss request from a remote cache targeting the same address, the cache line is transited to the IM_D_O state, which means the cache needs to respond to the read miss request and changes the line to the O state after it receives the data response.

We let the cache store the coherence steady states and allocate a 32-entry 2-way miss status handling register (MSHR) table to handle the transient states for cache misses and
write-backs. Each MSHR entry maintains a forwarding list to optimize the case where the cache receives a snoop request with the same address as a pending local write request that is still waiting for the data response. The MSHR records the snoop request in the forwarding list, allowing subsequent snoop requests to be served. The requests recorded in the forwarding list can be served as soon as the local write is completed.

### 4.1.5 Coherence Controller Microarchitecture

The coherence controller shown in Figure 4-4 serves as an ordering point and an interface to the next-level memory. The controller snoops, activates, and broadcasts every coherent
4.1. LEAP Coherent Memory

request as well as responds to the request if the target data block is not owned by one of the coherent caches. To improve performance, requests are allowed to be served out-of-order inside the controller before they get activated because the controller is the ordering point, which determines the global request order. The controller uses two LEAP private memories: one is the data memory, which serves as the interface to the next-level memory hierarchy; the other is the ownership memory, which stores the coherence ownership information per memory address.

As described in Section 4.1.2, the coherence controller has to ensure that each data block has exactly one owner, which responds to snoop requests, and to support automatic cache state transition from the I state to the M or O state on a read miss. To achieve this, the coherence ownership for each memory address is designed to include three states: the I (invalid) state indicating the next-level memory does not own the data, the E (exclusive) state indicating the memory has the exclusive copy of the data (and no cache owns the data), the O (owned) state indicating the memory owns the data but some of the caches may also have a valid shared copy. The controller responds to cache requests only if the target address is not in the I state. The controller grants exclusive ownership to an incoming read miss request if the target address is in the E state and grants nonexclusive ownership if the target address is in the O state.

To reduce the bit-width of the request rings, we separate data and ownership messages for write-backs and sending write-back data on the response ring. The controller contains a write-back status handling register (WSHR) table to track incomplete dirty write-back requests, allowing non-atomic dirty write-backs. As with the MSHRs at the client, the WSHR maintains forwarding lists. If there is a read miss request from one of the caches requesting the same address as an incomplete write-back, the WSHR records the read request and responds to the requester after the write-back data arrives.

4.1.6 Deadlock Freedom

Deadlocks are a classic problem of coherence protocols. We assume that gem5's MOSI protocol is deadlock free. To ensure deadlock-freedom in our implementation, we need to first guarantee that coherent caches do not stall due to insufficient buffer space in the
network. To achieve this, our implementation requires that the cache reserve request buffer slots inside its local router before it begins processing a local request. Similarly, an entry is reserved in the router's completion table for each activated request from the network before the request is processed. These reservation mechanisms guarantee that coherent caches can always consume incoming packets from the network. Since the coherent cache is never stalled, a response received from the network can always be consumed and hence does not block the messages from the response channel. At the network level, each message class is transmitted on an independent ring. No messages traverse the same channel twice, avoiding deadlocks in the logical network.

The logical network for coherent memories is implemented with latency-insensitive channels. We rely on the LEAP compiler described in Section 2.1.3 to implement a physical network for these channels, especially for the channels that cross FPGA boundaries. The network implementation produced by the compiler allocates virtual channels in a manner that guarantees deadlock freedom. The tools are also guaranteed to strictly increase channel buffering. Thus, our high-level buffer management is sufficient to guarantee correctness of the physical network implementation produced by the compiler.

4.1.7 Memory Consistency

To achieve high throughput, both the coherent cache and the coherence controller allow requests to be served out-of-order. These optimizations violate sequential consistency, which requires all memory operations to be executed in the order specified by the program. For programmers who need stronger memory consistency guarantees, as mentioned in Section 4.1.1, our coherent memory interface supports fences, including write fences, read fences and full fences, which can be invoked to ensure the ordering of memory operations. A write fence request is consumed by the local coherent cache after the cache sees all previously issued activated write miss requests. Before the fence is consumed, write requests issued after the fence wait in the request pipeline, while read requests can get processed. Similarly, a read fence blocks subsequent read requests and a full fence blocks all subsequent requests.
4.2  Coherent Cache Network Optimization

Although our coherence scheme automatically scales across FPGAs because coherent memories are connected via LEAP latency-insensitive channels, the memory architecture shown in Figure 4-2 is less efficient when the coherence domain spans multiple FPGAs due to long inter-FPGA latency. In addition, board-level memory resources on remote FPGAs such as DRAMs cannot be exploited when each coherence domain is managed by a single coherence controller, leaving memory bandwidth on the table.

To efficiently utilize memory resources on remote FPGAs, we introduce a memory interleaving technique for a coherence domain. The associated shared 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 4-5 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. To reduce the number of inter-FPGA traversals, coherent caches and interleaved controllers are connected via a hierarchy of ring networks.

Figure 4-5: LEAP coherent memories with dual coherence controllers.

4.2 Coherent Cache Network Optimization
Chapter 4. Shared Memory Abstraction

Within an interleaved coherence domain, each coherence controller is responsible for a portion of the domain address space. As the single-controller implementation described in Section 4.1.5, each controller connects to the next-level memory by instantiating two LEAP private memories 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 caches 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 single-controller implementation shown in Figure 4-2, to ensure all coherent memories see coherence requests in the same order, all requests must first go to the single, global ordering point before being broadcast in a global order, creating congestion at the ordering point. To alleviate congestion in 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, in the original implementation, coherent memory clients 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 and, as in the original implementation, fences can be used to enforce stronger consistency models.

To improve network performance, we partition the original ring network into hierarchical rings as shown in Figure 4-5. By default, coherent memory clients are partitioned
4.2. Coherent Cache Network Optimization

Figure 4-6: A walkthrough example of coherency management in a multi-controller system.

into equal-sized groups in lexical order, but other 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 inter-FPGA communication is reduced.
Chapter 4. Shared Memory Abstraction

Figure 4-6 shows a walkthrough example that demonstrates how we manage coherency in a multi-controller system and how the hierarchical ring network reduces the performance impact of long inter-FPGA latency. In the example, each FPGA has a coherence controller that is associated with two local coherent memory clients. Suppose client 0 is requesting block A, which is owned by client 1 at time T0 and belongs to the memory region managed by controller 0, and block B, which is owned by client 2 on the remote FPGA at T0 and belongs to the region managed by controller 1. At time T1, client 0 sends out a request for block A on the local unactivated request ring. At time T2, controller 0, which is the home controller for block A, receives the request and broadcasts it on the activated request rings, starting with local clients and then to the remote FPGA. At time T3, client 1 snoops the broadcast request A and sends out the response. At the same time, client 0 sends out another request for block B on the local unactivated request ring. Since the requested block is not local to controller 0, the controller transmits the request to controller 1, the home controller for block B, via the global unactivated request ring. At time T4, controller 1 receives and broadcasts the request. At time T5, client 2, the owner of the block, sends out the response, which traverses across FPGAs through the global response ring. The hierarchical ring network reduces inter-FPGA traversals when the requester, the owner, and the home controller of the target block (which is block A in the example) are all on the same FPGA. In addition, the broadcast request (request for block A) is first sent to the local requester (client 0) before it traverses across FPGAs, hiding the inter-FPGA latency from the local requester.

The optimization introduced in this section can be configured, enabled or disabled through parameters, which can be controlled either by programmers or compilers, customizing the coherent cache network for the target application and platform. For example, the number of interleaved controllers can be configured to match the number of board-level DRAMs across multiple FPGAs. Although this coherent cache network optimization is designed to improve the performance of shared memory applications mapped to multiple networked FPGAs, it can also benefit single-FPGA applications, especially when the FPGA platform has multiple board-level memories. The details will be discussed in Chapter 7.
4.3 Synchronization Primitives

In addition to memory coherency and consistency management, parallel programming in shared memory systems also requires synchronization, which is the coordination of multiple concurrent tasks to ensure correct access of the shared memory. Locks and barriers are common synchronization primitives. Locks limit access to shared resources, ensuring that no two accessors can enter a critical section at the same time. Only the accessor that obtains a lock has access to the associated shared resource. Barriers pause accessors until all accessors arrive at the barrier. When there are multiple processing engines, programmers need lock or barrier support if the shared tasks cannot be perfectly distributed or parallelized.

In general-purpose processors, inter-processor communication typically occurs through memory, and synchronization primitives are usually implemented using atomic memory operations. However, handling synchronization primitives through shared memory introduces coherence traffic, which is inherently expensive. Since FPGAs offer many communication mechanisms, we have more choices for the implementation of synchronization primitives. We implement lock and barrier services outside of the coherent memory framework. This section describes how we build these services and how to use them. For hardware-based applications, having separated synchronization services improves performance at low hardware cost: as we will show in Section 4.4.3, our synchronization primitive uses less than 4% of area compared to a coherent memory client.

4.3.1 Lock Service

Our lock service allows programmers to declare one or multiple locks shared among any number of clients. Clients access locks through lock nodes. We define a lock group as a group of lock nodes sharing single or a set of locks. Multiple lock groups can be instantiated using compile-time specified group names, similar to the way coherence domains are declared. To use the lock service, programmers need to instantiate lock nodes by specifying the name of the associated lock group. Although lock nodes have equal priority, we opt for a master-slave architecture to simplify the initialization of lock state.
at reset. Each lock group requires a master node. In the following example, four lock nodes sharing a lock are instantiated:

```plaintext
// Instantiate four lock nodes
// Lock group name: "IMG_A_LOCK", number of shared locks: 1
LOCK_IFC#(1) n0 <- mkLockNode("IMG_A_LOCK", MASTER);
LOCK_IFC#(1) n1 <- mkLockNode("IMG_A_LOCK", SLAVE);
LOCK_IFC#(1) n2 <- mkLockNode("IMG_A_LOCK", SLAVE);
LOCK_IFC#(1) n3 <- mkLockNode("IMG_A_LOCK", SLAVE);
```

The lock service provides an interface as follows:

```plaintext
interface LOCK_IFC#(type t_LOCK_ID);
    method void acquireLockReq(t_LOCK_ID id);
    method t_LOCK_ID lockResp();
    method void releaseLock(t_LOCK_ID id);
endinterface
```

A client calls `acquireLockReq` to attempt acquiring a lock. The client can only start accessing the shared resource until receiving the lock response (`lockResp`), which grants the client control of the lock. The client calls `releaseLock` when finishing the access.

Inside each lock group, lock nodes are connected via rings, and the master node of the group initially owns all the locks. Each lock node has a lock table that records a 2-bit lock state per lock. To improve performance, lock nodes also record lock forwarding information. Figure 4-7 walks through an example to demonstrate how the lock service works. In this example, four nodes form a lock group and share a single lock. At time T0, the master node (N0) owns the lock, and both nodes N2 and N3 issue a get lock request to the network. At time T1, N0 receives the request issued by N3 and sends out the lock as response, while N3 records N2 in its lock table. At time T2, N3 receives the lock and starts the critical section. At time T3, N3 finishes its task, releases the lock, and sends the lock to N2.

4.3.2 Barrier Service

Programmers use barriers to synchronize multiple concurrent tasks. Similar to lock groups, multiple barrier groups can be instantiated using compile-time specified group names. Our barrier service is centralized: each group of barrier nodes that share a single
4.3. Synchronization Primitives

or multiple barriers requires a master node. The master node is responsible for collecting barrier status from the slave nodes and broadcasting a completion signal. The following example shows how to instantiate three barrier nodes sharing one barrier:

```c
// Instantiate three barrier nodes
// Barrier group name: "IMG_B_BARRIER"
BARRIER_IFC n0 <- mkBarrierNode("IMG_B_BARRIER", MASTER);
BARRIER_IFC n1 <- mkBarrierNode("IMG_B_BARRIER", SLAVE);
BARRIER_IFC n2 <- mkBarrierNode("IMG_B_BARRIER", SLAVE);
```

The barrier service provides the following interface:

```c
interface BARRIER_IFC;
    method Bool initialized();
    method void setBarrier(t_BARRIER barrier);
    method void barrierReached();
    method void waitForSync();
endinterface
```

During initialization, the master node sets the synchronization condition (setBarrier), specifying which nodes must reach the barrier point before the barrier is completed. Then, the master broadcasts an initialization signal to inform all slave nodes that they can start
performing their tasks. When a slave node finishes (barrierReached), it sends a message to the master and waits for the barrier completion signal (waitForSync). The master node updates its barrier state upon receiving messages from slave nodes. When the barrier is completed, the master node broadcasts a completion signal, notifying the slave nodes that they can start their next step.

Our barrier implementation resembles the lock service implementation. The nodes that need to be synchronized are connected via rings to form a barrier group. The master node maintains a table tracking barrier states.

4.4 Evaluation

To evaluate the performance of our coherent memories and synchronization primitives, we target a set of benchmarks to both single and dual FPGA configurations. We use Xilinx VC707 FPGA boards as our evaluation platform. For dual FPGA configurations, we network the two boards using the high-speed SERDES provided by the FPGA fabric. If not specified, we use the following configuration to run the experiments:

1. Each LEAP private memory (PM) has a 1024-entry 64-bit direct-mapped cache.
2. Each LEAP coherent memory (CM) has a 1024-entry 64-bit direct-mapped cache.
3. Only one next-level cache is used in the dual FPGA tests.

4.4.1 Coherent Memory Service

To understand the overhead introduced by coherency management, we first measure the latency (Figure 4-8) and throughput (Figure 4-9) of systems with various number of coherent memories using synthetic benchmarks. The performance results of private memories are also included for comparison. Coherent memories have the same local cache hit latency as private memories, but their hit latency to the next-level cache is longer. This overhead results from the controller checking the ownership information of the requested data block and then forwarding the request to the ring connecting private memories (see Figure 4-2). In the dual FPGA configuration (Figure 4-8b), the measured
remote cache latency is higher than the next-level cache latency, because in this case the coherence protocol requires more traversals of the slow inter-FPGA link when data is cached remotely.

Figure 4-9 shows the throughput of various configurations of coherent caches. In the throughput tests, coherent memories access different regions in the shared address space, eliminating coherency traffic. The coherent memory’s write throughput is much lower than that of the private memory because coherent caches must first obtain data ownership before writes can be processed, which requires an extra transaction. The read throughput is similar to that of private memories, because the overhead of snooping is small.
Chapter 4. Shared Memory Abstraction

Figure 4-10: Performance comparison of different coherent memory configurations for *heat* with a various number of worker engines.

Figure 4-11: Performance comparison of coherent memories with single and dual coherence controllers when running *heat* at various frame sizes with 16 worker engines.

We also evaluate the performance of coherent memories on a real workload: a 2-dimensional heat transfer equation (*heat*) described in Section 2.3.1. *Heat* is a simple example of stencil programs, which have regular spatial locality and abundant parallelism and thus are often used as benchmarks to evaluate on-chip parallelism and the memory system performance.

To illustrate the value of coherent caching, we implement a second, uncached configuration where the coherent caches are replaced with an implementation that forwards clients' requests to a centralized, shared LEAP private memory. We refer to this configuration as the *uncached* configuration and the original coherent memory as the *cached*
4.4. Evaluation

configuration. We measure the execution time of running *beat* on a 512-by-512 grid of points for 128 time steps using the two configurations on both single and dual FPGA boards. In these experiments, each cache line can fit up to 8 grid points. To make a fair comparison, we scale the cache size of the LEAP private memory in the uncached configuration as the number of worker engines increases. In this way, the total cache size in the cached configuration is the same as that in the uncached configuration. Figure 4-10 shows the performance comparison of the two configurations. The performance speedup is calculated by normalizing the performance of each test to that of the single-worker baseline implemented with a single private memory. Coherent caches are successful in exploiting temporal and spatial locality, providing up to a $3.8 \times$ speedup when there are more worker engines, while the uncached configuration only gives a $1.4 \times$ speedup. The performance of coherent memories on dual FPGAs is worse than on a single FPGA, because inter-FPGA latency causes a higher miss penalty. In particular, the dual FPGA configuration is much slower when there are only two processing engines, because two engines do not generate enough pipelined traffic to cover the inter-FPGA latency.

To reduce the frequency of inter-FPGA traversals and to exploit board-level DRAMs on dual FPGAs, we optimize the coherent cache network by constructing an interleaved coherence controller for each FPGA to access its local DRAM through a private memory hierarchy. As mentioned in Section 4.2, the number of interleaved controllers can be configured through parameters. The coherent memory clients are evenly distributed among these interleaved controllers, and all the coherent memory components are automatically connected through hierarchical rings. Figure 4-11 shows the performance comparison of single and dual coherence controllers when running *beat* with 16 engines. The performance results are normalized to the single-worker implementation. Having interleaved coherence controllers on dual FPGAs provides around $2 \times$ performance speedup against the single-controller implementation. Performance improvement comes from doubled DRAM bandwidth and the cache network with higher scalability. When comparing with the single-worker baseline, performance gains are smaller for larger *beat* frame sizes because *beat* pixels are read in row-major order and evicted before they are reused when the frame size is too large.

65
4.4.2 Synchronization Service

To evaluate the efficiency of our synchronization services, we compare the performance of our lock and barrier primitives with two alternate implementations. One configuration implements locks and barriers using accesses to a set of coherent memories. To support through-memory software locks and barriers, we extend the coherent memory interface described in Figure 4-1 to include an atomic `testAndSet` operation. The other configuration uses a wire-based global arbiter to coordinate exclusive accesses and synchronization. Because this implementation does not communicate through latency-insensitive channels, it is limited to a single FPGA.

We test the three synchronization service implementations using a simple shared queue benchmark. In this test, a single queue is shared among several producers and consumers, which contend for the ownership of the tail and head of the queue. Upon obtaining the producer/consumer lock, the producer/consumer performs one operation and releases the lock. Table 4-1 shows the runtime of a 64-entry shared queue with two producers and two consumers. Each producer has 1024 items to insert. Compared to the wire-based implementation, our lock and barrier services introduce 20% overhead, while the implementation with through-memory software locks and barriers is much slower due to extra coherency transactions.
4.4. Evaluation

Table 4-3: FPGA resource utilization and maximum frequency of shared memory primitives.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>18K-bit BRAM</th>
<th>P&amp;R $f_{max}$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock Master</td>
<td>122</td>
<td>202</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>Lock Slave</td>
<td>81</td>
<td>176</td>
<td>0</td>
<td>383</td>
</tr>
<tr>
<td>Barrier Master</td>
<td>122</td>
<td>185</td>
<td>0</td>
<td>333</td>
</tr>
<tr>
<td>Barrier Slave</td>
<td>77</td>
<td>129</td>
<td>0</td>
<td>491</td>
</tr>
<tr>
<td>Coherence Controller</td>
<td>6795</td>
<td>7658</td>
<td>19</td>
<td>125</td>
</tr>
<tr>
<td>Coherent Memory Client</td>
<td>2985</td>
<td>5721</td>
<td>7</td>
<td>125</td>
</tr>
<tr>
<td>Private Memory Client</td>
<td>1660</td>
<td>2010</td>
<td>4</td>
<td>162</td>
</tr>
</tbody>
</table>

In addition to the shared queue benchmark, we also compare the throughput of our barrier service with a through-memory barrier implementation and an existing FPGA-based barrier system, which is implemented using mutexes in coherent soft processors [67]. The three barrier systems are evaluated with 1000 iterations on 8 threads. The through-memory barrier implementation runs at 110 MHz clock frequency, and the other two run at 125 MHz. Table 4-2 shows the barrier throughput comparison. The throughput of our hardware barrier primitive is $86 \times$ higher than that of the through-memory software barrier and $340 \times$ higher than that of the existing work. This demonstrates that it is much more efficient to build barriers outside of shared memory on FPGA. In addition, the through-memory barrier system using coherent memories achieves $4 \times$ throughput advantage compared to the existing system. The throughput difference may come from the overhead of running instructions on soft processors and the performance differences between the two coherent cache systems.

4.4.3 Implementation Area

Table 4-3$^1$ lists the maximum place-and-route frequencies and implementation areas of our shared memory primitives. The synchronization primitives require minimal area to implement. The coherent memory client uses approximately three times the slice LUTs of the baseline private memory. This size increase is not a surprise: coherence protocols

---

$^1$Memory primitives in the table target 64-bit data words with a 14-bit word address (an 8 KB cache of a 128 KB memory space). For comparison, the area of a LEAP private memory client addressing a similar region is also shown.
Table 4-4: Normalized area of coherent caches with various configurations.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>18K-bit BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>512-entry Cache</td>
<td>1.00</td>
<td>0.98</td>
<td>0.86</td>
</tr>
<tr>
<td>2048-entry Cache</td>
<td>1.00</td>
<td>1.11</td>
<td>1.57</td>
</tr>
<tr>
<td>4096-entry Cache</td>
<td>1.00</td>
<td>1.17</td>
<td>2.71</td>
</tr>
<tr>
<td>13-bit Addressing</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>15-bit Addressing</td>
<td>1.01</td>
<td>0.97</td>
<td>1.14</td>
</tr>
<tr>
<td>32-bit Data</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>128-bit Data†</td>
<td>1.41</td>
<td>1.55</td>
<td>1.71</td>
</tr>
</tbody>
</table>

† The coherent memory client's cache line size is changed to 128-bit.

are complicated. In addition to the extra slice resources, the coherent memory client uses more BRAMs as compared to the baseline client. The extra BRAM usage comes from the need to store coherence status bits in addition to the baseline cache metadata.

Our coherent memory clients as well as coherence controllers are highly parametric. Table 4-4 explores the effect of cache parameters on area cost. The area results are normalized to the area of the baseline coherent memory client shown in Table 4-3. Since most of the miss handling logic is constant regardless of cache size, increasing the cache size has very little impact on the area usage of the cache. Interestingly, BRAM usage does not scale linearly with cache size. This is because increasing the cache size decreases the cache metadata size such that entire cache lines can fit in a single BRAM word.

### 4.5 Summary

FPGAs have become interesting computational platforms. However, the infrastructure for programming FPGAs is still lacking, especially in the realm of shared memory parallel programing. To facilitate designing parallel algorithms on FPGAs, we extend the LEAP private memory abstraction and propose LEAP coherent memories, which provide a simple SRAM-like interface and automatically manages coherent caches. We also provide lock and barrier primitives, which leverage the native communications primitives of the FPGA rather than relying on shared memory. Because our primitives are framed in terms
4.5. Summary

of high-level latency-insensitive channels, they may be automatically partitioned across any configuration of FPGAs through LEAP compilation.

Unlike general-purpose processors that typically support only one global shared memory across all cores, LEAP coherent memories support multiple coherence domains and computation engines that do not need to access shared memory resources can simply use private memories. Programmers can customize FPGA memory systems on a per-application basis by declaring multiple disjoint address spaces, which can be either private or shared, avoiding unnecessary overhead introduced by coherence management. For example, in our joint work published in [95], private and shared memory regions for heap-manipulating HLS programs are identified through program analysis in order to construct application-specific memory hierarchies that contain both LEAP private and coherent memories.

This chapter describes our coherent memory interface, the first implementation of LEAP coherent memories and a coherent cache network optimization for better scalability. Because of the proposed shared memory abstraction, other coherence protocols and further performance optimizations can be applied under the abstraction in the future and will be transparent to the programs using LEAP coherent memories.
Both LEAP private memories and coherent memories provide simple, abstract memory interfaces that separate user programs from physical memory implementations. This separation permits a great deal of freedom in terms of optimization: system developers or compilers are free to choose any implementation that preserves the functional behavior of the abstract memory primitives without changing the application kernels. To automatically construct a memory hierarchy tailored for the target application, the second step is to perform optimizations under the memory abstractions in order to implement various memory building blocks from which an optimized memory hierarchy can be formed.

In this chapter, we examine several cache optimizations performed under LEAP memory abstractions to construct memory building blocks with different performance-cost tradeoffs, enabling better generation of program-optimized memory systems. Since many algorithm implementations do not fully utilize the resources available on the FPGA, the proposed optimizations leverage the spare resources to automatically speed up previously written FPGA applications expressed using LEAP memories. These optimizations do not need to provide a large average benefit across all potential applications (as is required in processors) because they can be applied only when they are able to improve the target program, avoiding unnecessary overhead.

We first present an FPGA-optimized cache prefetcher, the size of which is tunable to exploit excess resources. The design concepts and implementation details are included in Section 5.1. To consume substantial memory resources left by the user program,
Chapter 5. Cache Optimizations

we design a set of scalable on-chip caches in which additional cache access latency is traded for maintaining high frequency at large cache capacity. In addition, we extend LEAP’s memory hierarchy to include an on-chip shared cache that, for typical designs, can consume most unused BRAMs remaining after the construction of user kernels and other memory hierarchy components. This shared cache is built to reduce the miss latency of the first-level caches as well as to enable dynamic resource sharing among multiple memory requesters. In Section 5.2, we introduce the proposed scalable cache primitives and study cache design tradeoffs among access latency, operating frequency, and cache capacity for FPGA applications. To facilitate the exploration of application-specific cache microarchitecture and optimizations, we provide a framework that allows programmers to easily configure multiple levels of caches through parameters. For example, programmers can enable cache prefetching and cache banking, control cache size and associativity, or disable some levels of cache in the on-chip cache hierarchy if necessary.

We evaluate the effectiveness of our cache optimizations by applying them to several pre-existing FPGA programs. Our experiments show that adding prefetchers provides a performance gain of 17% on average, and up to 40%, on unmodified program source. In addition, our scalable cache microarchitecture achieves 7% to 74% performance improvement (with a 26% geometric mean), over the baseline cache microarchitecture when scaling the size of first-level caches to the maximum. Compared to the implementation with a minimal cache configuration, constructing maximal scalable caches achieves a $2.63 \times$ performance gain on average.

5.1 Cache Prefetching

5.1.1 Prefetching In FPGAs

Prefetching in general-purpose processors has been well studied and widely applied in modern architectures. However, there are significant differences between prefetching in general-purpose processors and prefetching in FPGAs, both in terms of program
5.1. Cache Prefetching

behavior and hardware implementation. By examining these qualitative differences we build intuition into our FPGA prefetcher design.

Hardware programs differ qualitatively from software programs. Hardware programs lack a PC, an important prefetching hint for general-purpose processors. Although software prefetchers have access to the PC, the memory access streams produced by software programs are a mix of both data and program control structures, which are usually non-predictable even in streaming applications. On the other hand, in FPGA programs, control structures, which must be stored in memory in Von Neumann architectures, are stored in the fabric and accesses to these structures do not pollute memory access streams. In addition, by using multiple, independent LEAP memories, FPGA data streams produced by accessing different data structures can often be separated. Therefore, the access stream observed by an FPGA prefetcher in a single LEAP memory is, in many cases, a pure data stream. Prefetchers in general-purpose processors rely on the PC to filter out non-predictable accesses, especially at the first-level cache. We show that reasonable prefetching is possible at the first-level cache in hardware programs even though the PC is not available.

In general-purpose processors, a common technique to improve program performance is software prefetching: the injection of extra load instructions into a program to pull useful data into the cache before its use. Hardware programs are generally described in a pipelined style, which superficially resembles software prefetching. In typical hardware implementations, memory requests are issued well in advance of data use, and the hardware pipeline is built to tolerate at least some, if not all, of the latency of these memory requests. Unlike software prefetching, which can slow a program with extra instructions, explicit program prefetching in hardware has almost no performance cost, beyond the introduction of new buffering. At first glance, explicit prefetching seems to obviate the need for architectural prefetching support in the FPGA. However, it is difficult for even well-designed legacy programs to anticipate the structure and behavior of new FPGA memory architectures or to completely hide the latency of the occasional long cache miss. We will demonstrate that even codes with well-engineered memory latency tolerance benefit from our prefetching architecture.
Chapter 5. Cache Optimizations

5.1.2 Prefetching Microarchitecture

Although FPGAs and general-purpose processors are different, the similarity between the memory hierarchy of LEAP memories and that of general-purpose processors allows
5.1. Cache Prefetching

us to borrow the concepts of prefetching techniques used in processors to improve the performance of existing FPGA programs. In this section we describe our adaptation of classical prefetching techniques to the computational structures of FPGAs and the integration of our prefetching microarchitecture into the LEAP memory hierarchy.

Since FPGAs lack a PC, we employ address-based stride prefetching, which separates global memory accesses according to their memory region, for LEAP private caches. The size of each memory region is set as the capacity of the private cache, which is programmer tunable but defaults to 8 kilowords. Because hardware programs are typically well-pipelined, memory requests can arrive temporally close together. Therefore, the prefetching learning process must be short enough to accommodate these back-to-back accesses, preventing us from using some of the more complicated prefetching schemes used in general-purpose processors. However, we believe that the address-based stride prefetching is sufficient to disambiguate multiple access streams because the memory accesses from hardware programs are pure data streams.

Our prefetcher is a variation of the classic addressed-based stride-prefetcher, to which we add more learning resources and automatic look-ahead distance adjustment. Memory accesses are separated into streams according to their memory regions, as denoted by the high-order bits of the memory access address. Each stream is directly-mapped to a learner in the prefetcher, permitting us to implement our prefetcher state storage in either FPGA-resource-efficient LUTRAM or BRAM. Learners in our prefetcher are updated by both cache misses and hits to prefetched cache lines, since each useful prefetch changes what would have been a cache miss into a prefetch hit.

Prefetch requests must be timely. If cache requests to a prefetched line can arrive before a prefetch has completed, the user cache request incurs some additional latency and the benefit of prefetching is reduced. On the other hand, if data is prefetched too early, it may evict other data that is still needed or get evicted before it is used. We attempt to adjust the prefetch look-ahead distance dynamically to match the timing of requests from the client hardware. When a client request stalls due to an outstanding prefetch to the same cache line or if a prefetch request targets a cache line that already has an outstanding request, we increment the look-ahead distance. The look-ahead distance may
be increased until it reaches a statically defined upper bound. To prevent the look-ahead distance from becoming unnecessary large, we add a negative feedback by decreasing the distance when receiving a certain amount of timely prefetch hits. One important feature of our prefetching architecture is portability: we expect that our prefetching scheme will be applied to multiple memory architectures and hierarchies. Dynamically adjusting the look-ahead distance is essential in this use case since different memory hierarchies and technologies will assuredly have different access latencies.

The main purpose of a prefetcher is to leverage the unused memory bandwidth to fetch data for future needs. When the memory bandwidth is already saturated by normal cache requests, prefetch requests consume the precious bandwidth resources and thus undesirably delay the responses to those normal requests. To prevent prefetching from overwhelming the memory bandwidth, the prefetcher automatically stops issuing requests (but keeps learning on cache accesses) when the number of outstanding memory accesses exceeds a statically defined threshold, which may vary for different memory hierarchies and technologies.

Our prefetch learners store five values, as depicted in Figure 5-1. Each learner stores a tag, the most recently referenced cache address from the associated stream (Prev Addr), the detected stride (Stride), which is the difference between the two most recent addresses, the look-ahead distance for the stream (Dist), and a 2-bit saturating counter for the prediction state (State). When a learner is triggered by a cache access (L) and has correctly predicted the stride (s) of that access, it issues a prefetch request $L + d \cdot s$ to the prefetch request queue, where $d$ is the look-ahead distance associated with the learner.

The left side of Figure 5-1 shows a modified LEAP private memory client augmented with prefetchers. The private cache is connected to a prefector that learns from the cache accesses and issues prefetch requests to bring data into the cache. The prefetcher consists of multiple learners, each of which is responsible for extracting the stride pattern in a local memory access stream. The number of learners is parameterized and can be adjusted either by the programmer or by a compiler depending on the amount of available resources. Because we store the state of the learners in either FPGA LUTRAM or BRAM resources, we can instantiate many more learners on an FPGA than in a
5.1. Cache Prefetching

processor, although these learners can only be accessed by direct address indexing. Indeed, because FPGA resources have a fixed size, we can fill the memory resources completely with learners – BRAM-based prefetchers may have thousands of learners. Using large numbers of learners approximates a fully associative structure and can significantly reduce conflict misses.

The right half of Figure 5-1 depicts the integration of our prefetching microarchitecture into the first-level private cache microarchitecture. In the baseline private cache microarchitecture, cache lines are either available for operation or are busy, if there is an outstanding request to the next-level memory. The baseline cache maintains two kinds of requests which are buffered in two distinct queues: one buffers incoming client requests (New Req) and the other stores prior requests that are blocked by busy cache lines (Retry Req). Each cycle, an arbiter chooses a single request from one of the queues. If the chosen incoming request needs to access a busy cache line, it is shunted to the retry queue where it waits for the busy line to be serviced. This allows subsequent cache requests to be processed out-of-order but requires the introduction of a completion buffer (not depicted) in the cache. Because out-of-order request processing obfuscates memory access patterns, our prefetcher learns only from the incoming client requests.

In addition to the prefetching hardware itself, we make a few modifications to the baseline cache microarchitecture. Our prefetching microarchitecture adds a third request source to the request arbiter: a queue that buffers requests issued by the prefetcher (Prefetch Req). If a prefetch request is chosen and the request tries to access a busy cache line, it is not shunted to the retry queue, but instead dropped. Prefetches to busy lines are dropped because it is likely that the prefetch request is issued too late, and the cache line is busy because the client request predicted by the prefetcher has already occurred. In addition, we add an additional prefetch status bit (PF) to each cache line to mark cache lines that are pulled into the cache by prefetch requests and have not been accessed yet. This additional bit enables the tagged prefetching scheme, in which a prefetch request is issued only when the referenced block is accessed for the first time, as well as allows the prefetching hardware to check whether a prefetch request is timely or not.
Table 5-1: Structural and performance metrics for board-level memories on FPGA evaluation platforms, as measured at the memory controller.

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Capacity</th>
<th>Bandwidth</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nallatech ACP</td>
<td>SRAM</td>
<td>8 MB</td>
<td>600 MB/s</td>
<td>150 ns</td>
</tr>
<tr>
<td>Xilinx ML605</td>
<td>DRAM</td>
<td>512 MB</td>
<td>2780 MB/s</td>
<td>254 ns</td>
</tr>
</tbody>
</table>

5.1.3 Evaluation

To evaluate the effect of adding our prefetcher to the LEAP memory hierarchy, we examine three previously published FPGA implementations: Blocked Matrix-Matrix Multiplication (MMM), Mergesort, and H.264 described in Section 2.3.1. These codes are several years old and were originally written targeting much smaller FPGAs than our evaluation platforms. As a result, adding our prefetching scheme does not materially impact the physical implementation of these designs in terms of area or maximum clock frequency. It is also important to note that these designs are highly performance-tuned: two of the designs (MMM [24] and Mergesort [33]) won performance-based FPGA design contests.

We run the three benchmarks on two different FPGA platforms: the Nallatech ACP module and the Xilinx ML605. The chief qualitative difference between the ACP and the ML605 is that the ACP board-level memory is a small, but fully-pipelined SRAM, while the ML605 board-level memory is a large DRAM, in which only some memory requests may be pipelined. Table 5-1 summarizes the properties of the board-level memories on the two platforms.

In this section, we first discuss the performance improvement achieved by our prefetcher for each workload. Then, we use MMM as an example to analyze the prefetching accuracy and the effect of controlling prefetch bandwidth. We also discuss the area of our prefetching hardware. If not specified, we use the following configurations throughout this section:

1. Each prefetcher has 32 learners.
2. The look-ahead distance is increased with an upper bound of 32 and decreased on every 4 timely prefetch hits.
5.1. Cache Prefetching

(3) Prefetch requests are ignored when the total number of outstanding memory requests exceeds 12.

5.1.3.1 Runtime Acceleration

**MMM:** We evaluate our prefetcher architecture using matrix multiplications of various sizes. For small matrix sizes, the prefetcher provides up to a 70% performance gain as shown in Figure 5-2a.

The underlying MMM hardware decouples memory operations from computation and actively attempts to overlap the latency of block loads with ongoing computation. When there are a sufficiently large number of blocks in a row or column, the hardware matrix multiplication hides most of the memory latency. However, at the edges of the matrix multiplication, for example starting on a new row of blocks, the MMM algorithm does not fully overlap loads and incurs some performance penalty due to memory latency. Our prefetching scheme minimizes this penalty, and in smaller problems, which are dominated by edge conditions, prefetching results in a substantial performance gain.

**Mergesort:** Mergesort is evaluated by sorting random lists of 128-bit integer values. Figure 5-2b shows that, our prefetching scheme marginally improves the performance of the sorter on both FPGAs.

Our prefetcher architecture is able to discover the list-based data management scheme that mergesort is using. However, as mergesort is intrinsically performing its own prefetching scheme by monitoring the fullness of its internal list buffers and preemptively requesting data, there is limited opportunity for performance improvement. Despite this, our prefetching scheme can mine out an additional 6% performance without requiring performance tuning on the part of the programmer.

**H.264:** Figure 5-2c shows the performance results of applying our prefetching scheme to an H.264 decoder operating on streams of various resolutions. Although the effect of prefetching varies among different video streams on different platforms, all benchmarks benefit from prefetching to some degree.
There are many factors that may cause the prefetching performance variation. First, because the encoding of a video is strongly dependent on the content of that video, the amount and kind of prefetching varies between encoded streams. Actually, for streams without inter-prediction, prefetching has no effect on performance.

In addition, at lower resolutions, data may remain in the cache for longer periods during processing and therefore can be reused, while at higher resolutions, only a small portion of a frame can remain on chip. As a result, useless prefetches that evict useful data in the cache may have bigger impact on lower resolution streams. However, the performance of higher resolution streams is more likely to suffer from limited memory bandwidth due to more outstanding requests.

The effect of prefetching on decode speed varies between the ACP and ML605. The ML605 has larger board-level memory latency, which means a timely prefetch results in a bigger performance gain. However, because the memory on the ML605 is not fully pipelined, useless prefetches waste memory bandwidth and may reduce performance.
5.1. Cache Prefetching

Figure 5-3: Prefetch result analysis for MMM.

5.1.3.2 Prefetch Accuracy

Figure 5-3a gives a classification of prefetches based on their dynamic behavior. Prefetches issued by our prefetcher microarchitecture fall into five categories. At issue, a prefetch may be dropped, either because the data already resides in the cache, or because the cache line target is busy, indicating that the prefetch is too late where the requested address may have been issued earlier by the client. Once issued, each prefetch has one of three possible outcomes: (1) timely useful, (2) late useful where the user program has already attempted to access the prefetch data, and (3) useless where the prefetch data is not accessed before it is evicted from the cache.

Although we show a detailed breakdown for only MMM, in general, prefetches are useful, that is they are either timely or late. Most programs that we tested also experience
some degree of improved runtime, though the runtime improvement may be small if there are few prefetches.

In the larger MMM workloads, a sizable portion of the prefetches are useless. In our blocked MMM implementation, we access half of the blocks in column-major order. Because of the prefetch look-ahead distance, the prefetcher fetches beyond the edge of the block into the next column. Although these prefetches will ultimately be useful, they are temporally distant and get evicted from the cache before they are used.

5.1.3.3 Prefetch Bandwidth Control

To prevent prefetching from overwhelming the memory bandwidth, our prefetcher automatically stops issuing requests when the number of outstanding memory requests exceeds a certain threshold, in effect permitting prefetching only if there is spare memory bandwidth available. Figure 5-3b and Figure 5-3c show the effects of controlling prefetch bandwidth on performance and prefetch issue rate. The low issue rate under bandwidth control indicates that MMM has high demand on the memory bandwidth, especially at large matrix sizes due to its own internal block prefetching. Without the bandwidth control, prefetch requests may overwhelm the memory and starve these memory requests, resulting in increased miss latency and runtime performance degradation. Bandwidth-limiting has a more pronounced effect on the ML605 because its board-level DRAM is not fully pipelined.

5.1.3.4 Prefetcher Area

Table 5-2 shows the area and frequency results obtained from building our prefetching hardware with two different configurations on the ML605 board. In general the area
5.2. Scalable Caches

requirements of prefetching in the FPGA are extremely small\textsuperscript{1}, even if there are many stream learners incorporated into the prefetch engine. This is because the learner state, the chief consumer of area in an ASIC prefetcher, can be mapped efficiently to on-chip SRAM, leaving only the much smaller tracking and address generation logic to be implemented in slice resources.

Most applications require only a handful of LEAP memories. Coupled with the area efficiency of our prefetcher implementation, this means that the performance gains of prefetching discussed above are not likely to compromise the overall implementation quality of most designs.

5.2 Scalable Caches

In general-purpose processors, architects usually need to fix the memory system parameters at design time. Parameters like the cache size, the number of ways, and the hierarchy of caches are chosen based on an expected set of workloads. For FPGAs, the situation is different. Memory system designers can choose a memory system per application and all the spare resources left by the user program can be consumed in the memory system to achieve higher performance. Although FPGA programmers have always had this freedom of choice, it is generally not used because a sufficiently rich set of primitives for memory system construction is not available and designing a memory system from scratch is too time consuming. Besides, how to construct a memory system to leverage extra resources in the most efficient way requires the exploration of a large design space. With LEAP memory abstractions, we can customize a memory system on behalf of the programmer without perturbing the user program. To facilitate memory system customization, we provide a richer set of memory primitives. This section describes our proposed scalable memory primitives and how we use them to construct memory systems that efficiently utilize spare BRAM resources to improve program performance.

To motivate our microarchitectural extensions, we develop a simple, classical performance model to estimate the performance of a target user program running on top of

\textsuperscript{1}The area used by a single prefetcher is less than .5% of the total area of the LX240T chip on ML605.
Chapter 5. Cache Optimizations

LEAP memories. In this model, we assume that the user program's computations are perfectly overlapped with memory operations, and the memory operations are serialized. The program performance is proportional to the system frequency over the maximum of two average latencies: the average computation latency and the average memory access latency, which is modeled by the hit latency of the first-level cache plus the product of the cache miss rate and its miss penalty.

The following formula describes this simple performance model:

\[ \text{performance} \propto \frac{f}{\max(n_c, n_{hit} + r \cdot n_{miss})} \]  

(5.1)

where \( f \) denotes the design frequency and \( r \) denotes the miss rate of the first-level cache. \( n_c, n_{hit}, \) and \( n_{miss} \) are the average latencies (in terms of cycles) for computation, first-level cache hit, and cache miss, respectively.

To improve program performance by utilizing extra BRAM resources left by the user program, we propose the following two approaches: (i) scaling the size of the first-level caches to decrease the miss rate \((r)\), and (ii) adding a large mid-level shared BRAM cache sitting on top of the off-chip cache to reduce the miss penalty \((n_{miss})\) of the first-level caches. We observe that naively scaling the size of caches usually leads to frequency degradation and thus may have negative impact on performance. We will discuss how we overcome this scalability issue in Section 5.2.2. In addition, to completely exploit BRAM resources, Section 5.2.3 describes how we construct caches with non-power-of-two sizes.

5.2.1 On-chip Shared Cache

In LEAP's baseline memory hierarchy, as shown in Figure 4-2, BRAM resources are consumed by the first-level caches in LEAP private memories and coherent memories. If there are multiple memory clients, partitioning and distributing on-chip memory optimally can be challenging. Consider a program containing different types of processing engines and each connected to a LEAP private memory. Constructing symmetric first-level caches may result in inefficient memory utilization, because each memory client may have a dynamically variable working set size or a different runtime memory footprint.
5.2. Scalable Caches

To improve the efficiency of memory utilization, we may prefer to use extra BRAM resources to build a shared cache, enabling dynamic resource sharing among multiple processing engines.

Figure 5-4 shows the extended private memory hierarchy with a shared BRAM cache sitting on top of the off-chip cache. Since LEAP coherent memories are built on top of LEAP private memories, shared-memory applications that use LEAP coherent memories can also benefit from this on-chip shared cache. Similar to the off-chip cache, each private memory space and shared memory domain is uniquely tagged and separated, so there is no need to handle coherence in the shared cache.

Unlike the baseline first-level cache, which is implemented as direct-mapped for simplicity, we design the on-chip shared cache to be set associative in order to reduce conflict misses. In the case that multiple LEAP memories share the memory hierarchy, a set associative cache with multiple ways can preserve multiple memory footprints simultaneously. The cache configuration parameters, including the number of sets and
Chapter 5. Cache Optimizations

the number of ways, can be controlled by either the programmer or by the compiler. Building caches on FPGAs is a complex balance of area and performance. While the baseline first-level cache has one word per cache line to reduce the area utilization of a replicated structure, the on-chip shared cache stores multiple words per cache line to take advantage of spatial locality and to better utilize backing store bandwidth, since theses stores usually have larger line sizes.

Figure 5-5 shows the microarchitecture of the set associative cache. Following the common design choices for building last-level caches in processors, we store metadata and actual data values separately and read them serially for power and timing optimization. The data store is divided into a set of BRAM stores, and each BRAM stores a single word for all cache lines, enabling faster word-sized writes. We implement a least-recently-used (LRU) replacement mechanism to select a victim way on a cache miss. When constructing a cache with high associativity, to maintain high frequency, a deeper pipeline is used to relieve the timing pressure introduced by tag comparisons, and BRAM banks of the cache data store are further divided into smaller banks, each storing a single word for cache lines from a particular way. To compensate the extra access latencies introduced by deeper
5.2. Scalable Caches

Pipelining, the cache can be optionally configured to read data values from the target cache set in parallel with metadata.

Since the control logic and the BRAM stores in the on-chip shared cache are separable, as shown in Figure 5-5, this cache structure can be built with our scalable BRAM store described in Section 5.2.2. Although we have introduced set-associativity as a primary mechanism for supporting shared caches, we note that this set associative structure can also be used in first-level private caches.

5.2.2 Cache Scalability

As Moore’s law has delivered more transistors, the amount of memory available on die has increased. Processor caches have grown larger and FPGAs have more BRAMs. Unlike general-purpose processors, where applications can make use of these new resources through the abstraction of the memory hierarchy, FPGA programs often cannot. This difficulty arises because FPGA programs are explicit in their use of memory: if a program asks for a physical 8 KB memory, there is little utility in scaling this memory to consume more BRAMs. However, even more abstract FPGA programs have difficulty in utilizing new memory resources in part because simply scaling up BRAM-based structures may have a negative impact on operating frequency and thus reduce overall design performance.

To improve BRAM scalability, we propose a multi-cycle banked BRAM structure, trading increased access latency for maintenance of frequency at higher capacity. Figure 5-6 shows the resulting banked BRAM microarchitecture. We split a large BRAM structure into multiple banks and add pipeline buffers at the input and output of each bank, relieving the timing pressure on cross-chip routing paths for requests and responses. A separate in-flight request queue is used to track the bank information for each request so that the responses from each bank can be reordered for return to the client.

Similar to the on-chip shared cache described in Section 5.2.1, the first-level caches in LEAP coherent and private memories are designed with separable cache control logic and BRAM stores. Therefore, we are free to replace the original BRAM stores in an existing cache implementation with the banked BRAM stores and form a new cache implementation with better scalability.
5.2.3 Non-power-of-two Caches

In general-purpose processors, it is common to build caches with power-of-two sets (for set-associative caches) or sizes (for direct-mapped caches) because of hardware implementation efficiency concerns. However, following the power-of-two sizing rules when constructing BRAM-based FPGA caches may cause inefficient resource utilization because the amount of BRAM resources on FPGA is fixed. As we will show in Section 5.2.6 and Table 5-3, for many applications we studied, around 35% of the available BRAMs are left unused when building first-level caches with maximal sizes under the power-of-two sizing limitations.

To completely utilize available BRAM resources when constructing on-chip caches, we need to eliminate power-of-two sizing restrictions. Constructing set-associative caches (as described in Section 5.2.1) with non-power-of-two ways is one option. However, set associative structures may not always be suitable for building fast first-level caches due to higher complexity introduced by multiple tag comparisons and replacement policy management as well as extra access latency introduced by deeper pipelining. To build fast, non-power-of-two, first-level caches, we instead modify the existing cache indexing mechanism for direct-mapped caches.
5.2. **Scalable Caches**

Figure 5-7: Cache indexing mechanisms for power-of-two and non-power-of-two caches.

**Figure 5-7** shows the two cache indexing mechanisms we use in power-of-two and non-power-of-two direct-mapped caches. To reduce the number of conflict misses introduced by power-of-two memory access strides, a classical bit hash function based on a cyclic redundancy check (CRC) is applied to the target address before the cache lookup is performed. We use CRC as the hashing function because it requires only exclusive-OR operations and thus is simple to implement. We choose an $m$-bit CRC function where $m$ is not smaller than the bit length of the cache address so that the CRC hash function performs a one-to-one mapping and is reversible. A reversible hash function is valuable for tag comparison to reduce storage. For power-of-two caches with $2^k$ cache lines, $k$ of the low-order bits in the hashed address are used as the cache index, while the rest of the bits are used as the tag. Since the hash function is reversible, only the cache tag needs to be stored in the cache as metadata and the original cache address can be recovered when performing cache write-back operations.

For non-power-of-two caches, we propose a simple non-power-of-two indexing scheme. Consider a non-power-of-two cache with $M \cdot 2^r$ cache lines where $M$ is an odd number and $M \cdot 2^r < 2^k$. We use $r$ of the low-order bits in the hashed address as the cache index.
base (IB) and the rest of the bits are used as the tag. Then, we take \( k - r + n \) of the low-order bits in the tag as the cache index range (IR) where \( n \) is a small fixed constant (\( n \) is fixed as 4 in our cache design). We take additional \( n \) bits to compute modulus for load-balancing purposes, preventing the case where most of the addresses get mapped to some of the cache ranges. Finally, the cache index is calculated as follows:

\[
\text{Index} = (\text{IR mod } M) \ll r + \text{IB}
\]  

(5.2)

To reduce the latency of cache index computation, the modulus results are stored in a pre-configured lookup table, which has \( 2^{k-r+n} \) entries. As a result, the cache index computation becomes simply concatenating the value obtained from a table lookup with the index obtained from a normal power-of-two indexing scheme. The target cache size is under constraints so that \( k - r \) is not larger than 4 and thus the lookup table would contain no more than 256 entries. Since our goal of constructing non-power-of-two caches is to improve resource utilization, having such sizing constraints would only limit the potential performance gains.

### 5.2.4 Design Tradeoff

As described in Equation 5.1, while having large caches can potentially increase the cache hit rate and improve program performance, the decrease in frequency or the increase in cache latency (if using the proposed banked BRAM structures) may cause performance degradation. Therefore, consuming all the available resources and building the largest caches that are physically possible may not be an optimal choice. Different programs may have different memory access and computation characteristics and thus have different sensitivity to changes in frequency, cache hit rate, and cache latency. Even if we have a wide variety of memory building blocks, picking an optimal memory is a challenging problem, and requires intense characterization of different memory parameterizations and topologies.
5.2. Scalable Caches

The following are four options to design caches based on the tradeoff among frequency, cache capacity and latency, where \( f_t \) is the target frequency and \( f_m \) is the achievable frequency in the memory system.

1. If \( f_m > f_t \), increase the cache capacity until it hits the frequency limitation.
2. If \( f_m \geq f_t \), adopt the banked BRAM structure with longer cache latency and then increase the cache capacity until it hits the frequency limitation.
3. If \( f_m \leq f_t \), decrease the frequency to \( f_t' \) (where \( f_t' \leq f_m \)) in exchange for larger cache capacity.
4. If \( f_m \leq f_t \), decrease the frequency to \( f_t'' \) (where \( f_t'' \leq f_m \)) and increase the cache latency (using banked BRAM) in exchange for larger cache capacity.

Option 1 preserves performance but has limited scalability. This method can be adopted for programs that do not benefit from large caches. Option 2 can be used for programs that are less sensitive to cache hit latency but more sensitive to cache capacity. For some cases where FPGA programs are highly sensitive to cache capacity, options 3 and 4 can be used if the performance gain from large caches is enough to compensate the loss due to frequency degradation. For user programs that need to run at a fixed clock frequency, options 1 and 2 are preferred, or multiple clock domains are required.

In Section 5.2.6, we will show that there is no single option that provides maximum performance improvement for all the benchmarks. Although it is not possible to make an optimal decision without sophisticated program analysis, having the compiler make greedy decisions may still provide performance gains over the baseline memory system.

5.2.5 Compile Time Optimization

To automatically utilize the BRAM resources that are not consumed by user designs, we design a compiler that decides whether to construct an on-chip shared cache as well as the capacity and associativity of the shared cache during compilation. We first run a cache parameter sweep study to build a database that stores the maximum achievable frequency and BRAM resource usage for each cache configuration, including the number of sets,
Chapter 5. Cache Optimizations

Figure 5-8: An example of a LEAP program built with the two-phase compilation flow.

the number of ways, and the type of BRAM storage. This database provides us with implementation feasibility information so that our program modifications minimize the impact on user operating frequency.

With the parameter database, we build target programs through a two-phase compilation flow. Figure 5-8 shows an example of a program using LEAP private memories and built with the two-phase compilation. Programs using LEAP coherent memories can be treated similarly. During the first phase of compilation, the user program and the memory hierarchy excluding the on-chip shared cache are built through RTL generation and synthesis. The memory hierarchy is constructed based on the user’s parameter choices, such as first-level cache sizes, or default settings if no choices have been made.

After the first phase, the compiler gathers the BRAM usage information from RTL synthesis. If the amount of remaining BRAM resources is sufficient to construct an on-chip shared cache with a reasonable size, the compiler will use the pre-built database to greedily select the largest cache that can be implemented while using the remaining

92
5.2. Scalable Caches

BRAMs at the target frequency. If there are multiple candidates with the same cache capacity, the one with the highest associativity will be chosen. During the second-phase of compilation, the compiler then constructs the selected on-chip shared cache and the connections to the rest of the memory hierarchy. On the other hand, if there are no sufficient BRAM resources left, the compiler will directly connect the private memory controller with the central cache controller, which manages the accesses to the off-chip memory. Finally, the compiler passes the design to a standard FPGA tool flow to produce the final FPGA implementation while reusing most of the synthesis results obtained in the first phase.

5.2.6 Evaluation

We evaluate our scalable memory primitives on the Xilinx VC707 platform. Our VC707 deployment includes a 1 GB DDR3 memory, which we use to implement the off-chip cache. We examine a set of benchmarks ranging from kernels to large programs that nearly fill the VC707: memperf, heat, HAsim, merger, prio, and filter, as described in Section 2.3. The HAsim results in this section are obtained by scaling the first-level cache in its largest private memory, which is used to capture the virtual memory state of the modeled processors. Our filter implementation splits the tree into eight independently-processed sub-trees and uses 24 LEAP private memories: eight each for the sub-tree nodes, stacks, and candidate center sets.

For HLS benchmarks (merger, prio, and filter), we utilize Vivado HLS. We make use of Synplify 2014.03 for synthesis work and Xilinx Vivado 2015.1 for rest physical implementation work. All resource utilization and clock rate results in this section are post-place-and-route results.

5.2.6.1 BRAM Utilization

One of the central premises of this work is that many FPGA programs do not make use of all available on-die FPGA memory resources. Table 5-3a lists the BRAM utilization for the various benchmarks examined in this study. For each application, we report the
Chapter 5. Cache Optimizations

Table 5-3: Post-place-and-route results for applications with different cache sizes.

(a) BRAM utilization\(^\dagger\)

<table>
<thead>
<tr>
<th>Program</th>
<th>User-Kernel BRAM Usage (%)</th>
<th>Total BRAM Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>Memperf</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>HASim (16 cores)</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Heat (1 worker)</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Heat (8 workers)</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Merger</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>Prio</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>Filter</td>
<td>7</td>
<td>18</td>
</tr>
</tbody>
</table>

(b) Maximum achievable frequency

<table>
<thead>
<tr>
<th>Program</th>
<th>Post P&amp;R (f_{\text{max}}) (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
</tr>
<tr>
<td>Memperf</td>
<td>150</td>
</tr>
<tr>
<td>HASim (16 cores)</td>
<td>110</td>
</tr>
<tr>
<td>Heat (1 worker)</td>
<td>150</td>
</tr>
<tr>
<td>Heat (8 workers)</td>
<td>125</td>
</tr>
<tr>
<td>Merger</td>
<td>140</td>
</tr>
<tr>
<td>Prio</td>
<td>140</td>
</tr>
<tr>
<td>Filter</td>
<td>140</td>
</tr>
</tbody>
</table>

\(^\dagger\) BRAM utilization is reported as a fraction of the total BRAMs on VC707.
\(^\ddagger\) \(\max (2^k)\) denotes the maximal cache size limited by the power-of-two sizing rule.

user-kernel BRAM usage to give an idea of the amount of BRAMs available for platform optimizations. We also report the total area utilization when building with minimal and maximal first-level cache size configurations. Generally, the BRAM utilization for the user program is quite low: most of the applications that we study are computational kernels. Even \(\text{HAsim}\), which is a heavy consumer of logic resources for larger core-count models, uses relatively few BRAM resources. Abundant, unused resources permit us to build very large memory systems on behalf of the user program. For each application we studied, we could use more than 50% of the available BRAMs in support of the memory hierarchy. Under the power-of-two cache sizing limitation, there are still some BRAMs left after the
5.2. **Scalable Caches**

![Graph showing L1 cache performance with various microarchitectures.](image)

Figure 5-9: L1 cache performance with various microarchitectures.

Caches are scaled to the maximal sizes, especially for single-memory applications, such as *memperf*, single-worker *beat*, and *prio*. Without the cache sizing limitation, we can utilize more than 90% of the BRAMs for many applications.

### 5.2.6.2 Large First-Level Private Caches (L1 Caches)

In order to make use of the BRAM resources available on a large FPGA, we need to construct large caches. Figure 5-9 shows a study of cache size and microarchitecture using *memperf* with the stride equal to one and with the working set equivalent to cache capacity. In this figure, the throughput of *memperf* is reported for various cache implementations running at frequencies ranging from 75 MHz to 150 MHz, and flat lines indicate a failure to meet timing at a given target frequency.

The chief advantage of our banked cache microarchitecture is its ability to scale to very large capacities while largely maintaining frequency. As shown in Figure 5-9, a monolithic 2 MB cache, which uses more than 60% of the BRAM available on the VC707, runs at 75 MHz, while a 4-way banked cache can achieve around 1.8× of that frequency. The timing relaxation afforded by our buffered architecture enables individual banks to clock faster than similarly sized monolithic caches.

In theory, all of our caches running at the same frequency should have the same performance for *memperf*. In practice, cache performance is variable within a small range.
Chapter 5. Cache Optimizations

Because we use a bit-hashing scheme to reduce the number of cache lookup conflicts and to improve overall cache performance, the number of conflict misses is nearly uniform but varies with working set size. Banked caches have slightly lower performance than monolithic caches at frequency and capacity parity, due to the pipeline latency in the banked architecture.

The preservation of user operating frequency is a key challenge in implementing large memory systems. Figure 5-9 shows that we can maintain frequency for a small kernel. In general, our large banked caches do not incur a significant frequency penalty. Table 5-3b lists the operating frequency for each application with minimal and maximal cache size configurations. The table includes the frequencies of the maximal memory system configurations implemented with monolithic and banked BRAM stores. With banked stores, even when using most of the on-chip BRAM resources, we suffer a frequency penalty no larger than 20%. Compared to monolithic BRAM stores, banked BRAM stores achieve up to 100% frequency gain when building large caches.

5.2.6.3 Application Performance with Large L1 Caches

To evaluate the performance impact of building large first-level caches, for each application, we build the program with various cache sizes and microarchitecture configurations and compare the runtime performance when executing the program at its maximal achievable frequency. We will show that memory configurations with maximal cache sizes do not always deliver the best performance.

HAsim: Intuitively, a processor model should obtain performance gains from larger caches, just as processors do. Figure 5-10b bears this intuition out: the number of HAsim’s cache misses drops dramatically as we scale the cache size and capture larger portions of the model working set. However, this improvement in the cache miss rate does not translate into absolute performance. Rather, HAsim’s performance with respect to the cache size rises until the cache reaches a medium size and then plateaus, rising by a
5.2. **Scalable Caches**

maximum of 4%, as shown in Figure 5-10. This limited gain is a result of the deep pipelining of the *HA* model. Once a small first-level cache filters enough requests to reduce the bandwidth of requests to backing caches, *HA* is able to tolerate round-trip latency to DRAM without loss of performance. For less well-pipelined systems, like soft processors, the performance gains from our approach would be larger.

Figure 5-10: Performance metrics for *HA* with various cache configurations.

---

*HA* performance drops as simulated core count grows because all cores are simulated in a shared, multiplexed pipeline.
Because \textit{HA sim} is latency tolerant, trading any frequency for cache capacity is a loss on the VC707 (thus options 1 and 2 in Section 5.2.4 are preferred). \textit{HA sim}'s maximum operating frequency tops out at 110 MHz, enabling us to use our largest banked cache. Thus, although our banked caches do not help \textit{HA sim} much, they do not harm it either. On the other hand, large, monolithic caches degrade \textit{HA sim}'s performance by up to 30%.

\textbf{Heat:} Figure 5-11 shows the performance of single-worker and multi-worker \textit{heat} with various cache configurations and problem sizes. Generally, the performance of the \textit{heat} stencil is determined by the block size that can be fit into cache. If the problem size is too large to fit into cache, then \textit{heat} will always miss to the nearest memory sufficient to hold its working set. As we scale our caches, the larger \textit{heat} problems see large performance improvements. In the single-worker implementation, when building a very large cache, our banked microarchitecture outperforms the baseline monolithic cache by 74\% on average across a set of problem sizes, due to high operating frequency. In the 8-worker implementation, since the maximal cache size is much smaller compared to that of the single-worker version, the frequency penalty of monolithic microarchitecture is lower, causing a smaller performance difference between banked and monolithic caches. With our banked microarchitecture, the maximum frequency of single-worker \textit{heat} declines less than 10\%, even when implementing a two megabyte cache. Due to complicated data paths in coherent caches, multi-worker \textit{heat} runs at slightly lower frequencies and the frequency penalties of large caches are higher. \textit{Heat} is fairly sensitive to memory latency: when building small caches, monolithic caches slightly outperform banked caches. For smaller problem sizes, having smaller but fast caches is preferred (option 1 in Section 5.2.4); for large problem sizes, having large banked caches and slightly downgrading the frequency (option 4 in Section 5.2.4) provides the best performance gain.

\textbf{HLS Kernels:} In Figure 5-12, we explore scaling the capacity of the L1 caches for our HLS kernels. These applications use BRAMs internally, but use less than 20\% of the resources on chip. Our cache structures enable these applications to gain some benefit from the remaining on-chip resources. Using our approach, \textit{prio} is able to use 96\% of the BRAM resources.
5.2. Scalable Caches

Figure 5-11: Performance comparison for *heat* with various cache configurations.

All of our HLS kernel applications involve pointer chasing, and are, therefore, sensitive to memory latency. Increased capacity helps these kernels to the extent that they avoid long latency misses. For example, *merger* obtains a $5.5 \times$ performance improvement as its cache scales. At the same time, larger cache latency results in lower performance. With
merger, for small cache sizes, banking results in performance degradation due to increased latency. However, as the cache size scales, the superior operating frequency of the banked cache results in a 23% absolute performance gain over the monolithic cache. In addition, without the power-of-two cache sizing limitation, the performance improvement of prio increases from $1.75 \times$ to $2.39 \times$ with BRAM utilization increased from 80% to 96%. Filter obtains a smaller performance gain compared to other applications because it uses many LEAP memories. Each L1 cache in filter has a smaller capacity scaling range as we evenly distribute BRAM resources to all L1 caches in the program. In addition, since only the working set of tree nodes is large, only these memories benefit from cache capacity scaling. Since our HLS kernels are sensitive to cache capacity, the cycle performance gain is enough to cover the loss due to frequency degradation (option 4 in Section 5.2.4).

Summary: Figure 5-13 summarizes the performance improvement achieved by building largest L1 caches with the banked BRAM structure. For each benchmark, the banked cache performance is normalized to that of the monolithic cache with the same cache size. When constructing L1 caches with maximal sizes, our banked cache, which runs at higher operating frequencies, provides 7% to 74% performance gains (with a 26% geometric mean) over the baseline monolithic cache.

1 A geometric mean is used for HA sim and heat, which have multiple problem size configurations.
5.2. Scalable Caches

Figure 5-13: Performance comparison of the banked cache against the monolithic cache for various benchmarks built with maximal L1 cache sizes.

Figure 5-14: Implementation space of L2 caches.

5.2.6.4 Constructing Large On-Chip Shared Caches (L2 Caches)

In optimizing application memory systems, our goal is to consume all unused BRAM resources without negatively impacting program frequency. We therefore introduce a second-level on-chip shared cache into our memory hierarchy and then size it to achieve the user frequency target (Options 1 and 2 in Section 5.2.4). Figure 5-14 describes how we choose the size of this L2 cache. By exploiting set associativity, we are able to cover a broad
Chapter 5. Cache Optimizations

Figure 5-15: Relative throughput gain for memperf.

swath of the frequency-utilization space. Like our large L1 caches, our set-associative caches obtain frequency scalability by using our banked storage element.

Figure 5-15a shows the result of running our L2 cache selection algorithm on memperf. For this benchmark, our algorithm selects a 4-way cache with 8192 sets, utilizing about 33% of the BRAM resources available on chip. For those working sets that fit in the L2 cache, performance improves by about 25%. For completeness, in Figure 5-15b, we compare our algorithm against a direct mapped L1 with equal area (128K words). The large L1 handily outperforms our L2 for those sizes which fit in the L1. This is not a complete surprise, since in a single-memory-user case, the latency penalty of L2 can only lower performance. We imagine that in situations with more diversity of memory use, the L2 will be more beneficial. Interestingly, the L2 implementation outperforms the L1 implementation for working sets which do not fit in the cache. This is because the L2 captures some spatial locality within the stride sets.

Table 5-4 shows the result of introducing an on-chip L2 cache in merger. When the L1 cache size is fixed, adding an L2 cache always improves performance, although merger prefers large private caches to a large shared cache. Yet, even when building largest feasible L1 caches, there remains 25% unused BRAM resources, which can be used to implement a 4-way L2 cache with 4096 sets, improving performance by 6.6%. The performance gain
5.2. Scalable Caches

Table 5-4: L2 cache performance gain for merger.

<table>
<thead>
<tr>
<th>Total L1 Cache Size (KB)</th>
<th>BRAM Usage (%)</th>
<th>Runtime (s)</th>
<th>Performance Gain (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Only</td>
<td>L1 + L2</td>
<td>L1 Only</td>
</tr>
<tr>
<td>32</td>
<td>18.7</td>
<td>52.3</td>
<td>803.8</td>
</tr>
<tr>
<td>512</td>
<td>32.5</td>
<td>64.5</td>
<td>642.1</td>
</tr>
<tr>
<td>1024</td>
<td>46.8</td>
<td>80.8</td>
<td>433.2</td>
</tr>
<tr>
<td>2048</td>
<td>75.1</td>
<td>93.3</td>
<td>145.9</td>
</tr>
</tbody>
</table>

comes from higher associativity and multi-word cache lines that capture a degree of spatial locality in the L1 miss stream.

Although the inclusion of a shared L2 cache does benefit real programs, our results suggest that allocating resources to a large shared cache is less beneficial than allocating them to large L1 caches, at least for the applications we have studied. In general-purpose processors, where the memory implementation is fixed, caches cannot be too large because this negatively impacts frequency, slowing all applications, whether they benefit from the cache or not. On the other hand, the FPGA memory system can be customized with large L1 caches for specific applications, and the decision to trade frequency for capacity and latency can be made on a case by case basis. For many well-pipelined, bandwidth-intensive applications, a shared cache, even if implemented with otherwise unutilized resources, may have limited performance benefit. However, we believe that FPGA applications that have multiple memory requesters with dynamically variable working set sizes such as object tracking/labeling algorithms and multi-core soft processors can benefit from having a large shared cache. A future work is to explore these applications.

5.2.6.5 Energy Consumption

Our goal of designing scalable on-chip caches is to utilize spare BRAM resources for constructing application-specific cache hierarchies that are optimized for minimum program execution latency. In Section 5.2.6.3 and Section 5.2.6.4, we have shown that we can achieve up to a 5.9× speedup by constructing scalable caches. However, these performance-optimized solutions may not always achieve the best energy efficiency.
Table 5-5: Power and energy measurements for HLS applications†.

<table>
<thead>
<tr>
<th>Total L1 Cache Size (KB)</th>
<th>Performance Speedup</th>
<th>$P_{DRAM}$ (W)</th>
<th>$P_{FPGA}$ (W)</th>
<th>$E_{TOTAL}$ (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Merger</strong> (Input: 4*79990 random non-negative integers)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.00</td>
<td>1.75</td>
<td>1.89</td>
<td>2926</td>
</tr>
<tr>
<td>512</td>
<td>1.25</td>
<td>1.75</td>
<td>2.36</td>
<td>2643</td>
</tr>
<tr>
<td>1024</td>
<td>1.86</td>
<td>1.55</td>
<td>2.67</td>
<td>1827</td>
</tr>
<tr>
<td>2048</td>
<td>5.51</td>
<td>1.03</td>
<td>3.84</td>
<td>711</td>
</tr>
<tr>
<td><strong>Prio</strong> (Input: 327676 random non-negative integers)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.00</td>
<td>1.18</td>
<td>1.74</td>
<td>37,074</td>
</tr>
<tr>
<td>512</td>
<td>1.18</td>
<td>1.16</td>
<td>2.25</td>
<td>36,660</td>
</tr>
<tr>
<td>2560</td>
<td>2.39</td>
<td>1.01</td>
<td>3.53</td>
<td>24,134</td>
</tr>
<tr>
<td><strong>Filter</strong> (32759 tree nodes, 128 clusters, 256 iterations)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>1.00</td>
<td>1.00</td>
<td>2.38</td>
<td>54</td>
</tr>
<tr>
<td>768</td>
<td>1.05</td>
<td>1.01</td>
<td>2.84</td>
<td>59</td>
</tr>
<tr>
<td>1536</td>
<td>1.08</td>
<td>1.02</td>
<td>3.83</td>
<td>71</td>
</tr>
<tr>
<td>3072</td>
<td>1.40</td>
<td>1.01</td>
<td>5.17</td>
<td>70</td>
</tr>
</tbody>
</table>

† For each cache size, we select the cache microarchitecture (either monolithic or banked) that achieves the better runtime performance and report the performance, power, and energy measurements.

To evaluate the impact of our scalable caches on power and energy efficiency, we measure the power consumption on the FPGA and on the board-level DRAM using the Fusion Digital Power Designer package provided by Texas Instruments (TI). Table 5-5 shows the average FPGA and DRAM power consumption ($P_{FPGA}$ and $P_{DRAM}$) as well as the total energy consumption ($E_{TOTAL}$) for each HLS benchmark with various first-level cache sizes. The table also includes the performance speedup against the performance of the program with a minimal cache size. As shown in Table 5-5, scaling up first-level caches always increases the power consumption on the FPGA. For merger and prio, large caches effectively reduce the DRAM power, and the performance gains are large enough to compensate the increase in the FPGA power, resulting in higher energy efficiency. For filter, on the other hand, although large caches provide better runtime performance, they are less energy efficient compared to small caches. For applications that have energy constraints, the tradeoff between performance and energy needs to be considered when constructing the cache hierarchy.
5.3 Summary

Some of the optimizations and experiments in this section were conducted in collaboration with Elliott Fleming, Felix Winterstein, and Michael Adler. The indexing mechanism for non-power-of-two caches and the two-phase compilation flow for automatic L2 cache insertion were designed primarily by Elliott. Michael helped refining the timing paths in the banked cache microarchitecture and measuring HAsim performance. Most of the performance and energy measurements for HLS applications were conducted by Felix.

5.3 Summary

Programs expressed using LEAP memory abstractions provide a lot of freedom in choosing memory system implementations and optimizations. We have demonstrated that it is possible to exploit unutilized resources to construct memory hierarchies that accelerate the user program without design changes. The space of potential memory hierarchies is large and the problem of deciding how to build application-optimized memory hierarchies is difficult. In this chapter, we provide a large set of memory building blocks, including caches with prefetching logic and scalable caches, and a framework for integrating them, enabling experimentation in both of these spaces.

We extend the LEAP memory configuration interface, allowing programmers to manipulate various parameters in order to configure a multi-level on-chip cache hierarchy, enable or disable a certain cache optimization, and explore design tradeoffs for both private and shared on-chip caches. Most of the cache optimizations introduced in this chapter currently require manual configuration to achieve high efficiency: programmers need to manually set the optimization parameters in order to customize the memory hierarchy for the target application and platform. However, the proposed memory configuration framework also facilitates automatic construction of program-optimized memories: compilers driven with synthesis and behavioral feedback from a specific program can automatically apply cache optimizations through configuration parameters. For example, our joint work published in [96] estimates the cache performance of HLS applications through trace-driven program analysis and leverages the configuration
framework to automatically customize the sizes of first-level caches forming an application-specific multi-cache system with a maximum aggregate hit rate given available memory resources. In Chapter 7, we will also discuss some other automatic optimizations enabled by a feedback-directed compiler. A future work is to develop more sophisticated program analysis and automate the optimizations introduced in this chapter for a more general class of FPGA applications.
Service Communication Abstraction

The optimizations we examine in Chapter 5 focus on the microarchitecture of on-chip caches in order to achieve higher cache bandwidth and hit rate. However, in addition to the cache performance, the performance of a multi-level memory hierarchy may also be affected by the memory network connecting different levels of caches. As FPGA deployments move towards design entry points that are more serial, such as C-based kernels compiled through HLS, memory latency has become a serious design consideration. Though parallel, HLS programs are sometimes less parallel than conventional designs written in RTL, making them more sensitive to latency in the memory subsystem. As mentioned in Section 2.1.1, LEAP memories are by default connected through unidirectional latency-insensitive rings because rings have low complexity and are easy to assemble even when the number of clients to be connected is unknown prior to compilation. When the number of memory clients in a memory system scales (for supporting applications with a large number of processing engines), the long network latency introduced by ring-based memory networks may negatively impact memory latency and thus the overall program performance. For those applications that have a large number of memory clients and are sensitive to memory latency, to achieve high performance, it is essential to optimize memory networks by constructing more complicated network topologies with better scalability.

Since applications usually have different memory access behavior, similar to cache microarchitecture optimizations, the memory network also needs to be customized for
each target application in order to achieve high efficiency. This is especially important when the memory clients of the target program have asymmetric memory access characteristics. 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. This memory network customization is more challenging than configuring previously proposed cache optimizations, such as deciding cache sizes or banking options, because the design space of network topologies is much larger. Due to the large design space, manual exploration is unattractive, and an automated solution is desirable.

We believe that a memory abstraction is one of the key factors for automating the construction of application-specific memory systems. Similarly, we need a communication abstraction to facilitate the design space exploration of memory networks. We 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. In addition to memory networks, this abstraction can also be used for other centralized services that have demanding performance requirements. This chapter describes the proposed abstraction and introduces the network topologies that can be automatically constructed by our compiler. The details of our compiler design and how it determines an optimal memory network will be covered in Chapter 7.

6.1 Motivating Example

The advantage of customized memory networks is most salient for applications with a large number of asymmetric memory clients. One example is the *filter* application described in Section 2.3.2. In our case study, *filter* is divided into $P = 8$ independent subtrees. Each partition, which can be processed in parallel, uses 3 LEAP private memories to store 3 data structures: a sub-tree, a stack, and candidate centers, resulting in 24 LEAP
memories. The accesses to these data structures are essentially pointer chasing, which makes the execution time of filter very sensitive to the memory access latency.

In the baseline LEAP private memory hierarchy as shown in Figure 2-3, all 24 memory clients are connected on a single ring, introducing long network latency. To improve performance we need to build a cache network with better scalability. In addition, we observe that the memory clients in filter 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 7.3, we will show that our cache network optimized for filter provides a 44% performance gain over the baseline LEAP memory hierarchy.

6.2 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 implementations. 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. Similar to other LEAP communication primitives, such as point-to-point named channels and broadcast rings described in Section 2.1.1, this network abstraction is also latency-insensitive. While in this thesis we use this abstraction for memory services, it can also be used for other services that require high-performance network implementations.
Chapter 6. Service Communication Abstraction

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 6-1: The abstract interfaces of service connections.

The service connection abstraction provides clients and servers with request-response-based interfaces as shown in Figure 6-1. The abstract interfaces allow compilers to construct various network topologies underneath without changing the rest of the system, much the same as LEAP abstract memory interfaces enable the construction of different memory system implementations without changing the user program. This abstraction is an improvement over the LEAP ring primitives as described in Section 2.1.1 because it does not place the burden of passing other client’s traffic on each client and therefore does not presuppose a topology.

Figure 6-2 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 matching 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.

This network construction strategy can also be applied to services with multiple servers. For example, multiple memory controllers may be instantiated to manage accesses to multiple on-board memories, which we will discuss in detail in Chapter 7. In
6.3. 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 6-3. Figure 6-3a 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. Each client is connected to a ring node. 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 6-3b shows an example of a hierarchical-ring network, which consists of multiple levels of rings connected by 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 6-3 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outer forwards requests from its 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 6-1
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\(^1\). 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

\(^1\)The arbiter was designed primarily by Elliott Fleming. I optimized it by adding bandwidth upper limits.
6.3. Network Topologies

Algorithm 6-1 Arbiter with bandwidth control.

```plaintext
1: procedure REQUEST_SCHEDULING(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, ..., 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:       end
18:     end for
19:     if activeChildren ∩ hungryChildren ≠ ∅ then
20:       candidates ← activeChildren ∩ hungryChildren
21:     else if activeChildren ∩ priorityChildren ≠ ∅ then
22:       candidates ← activeChildren ∩ priorityChildren
23:     end
24:     ▷ Select the winner from candidates using round-robin
25:     winner ← ROUNDROBIN(candidates)
26:     Forward a request from winner to the output port
27:     for i = 1, 2, ..., LENGTH(childList) do
28:       if childList[i] is winner then
29:          histList[i] ← histList[i] << 1 + 1
30:       else
31:          histList[i] ← histList[i] << 1
32:     end for
33:   end while
34: end procedure
```

responsible for forwarding responses from the parent node to its children based on the requester ID tagged with the response. Similar to the hierarchical-ring network, to make response forwarding easier, the ID of each leaf node is assigned in a sequential order.

The three kinds of network topologies shown in Figure 6-3 implement different cost-performance tradeoffs. The single-ring network has low design complexity but introduces long network latency when there are many 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 client with few outstanding memory requests 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. The next chapter will describe how we enable the automation with feedback-directed compilation.
The key steps to automate the construction of application-specific memory systems, as we argue in Chapter 1, are defining abstractions, performing optimizations under the abstraction, and developing a resource-aware, feedback-driven optimization process. Armed with the memory and communication abstractions introduced in Section 2.1.2, 4.1 and 6.2, as well as configurable caches and memory networks developed in Section 5.1, 5.2 and 6.3, the final step is to develop a build procedure that automatically selects the most suitable building blocks and implements a memory hierarchy optimized for the target application. In this chapter, we present the LEAP Memory Compiler (LMC), which extends the LEAP compilation flow described in Section 2.1.3, to construct a program-optimized memory hierarchy based on the target program’s memory access behavior and the available memory resources on the target platform.

LMC is a resource-aware, feedback-driven compiler that operates in three phases: instrumentation, analysis, and synthesis. In the first phase, LMC constructs a dynamically-configurable network profiler with program instrumentation logic inserted at each memory client in order to measure the client’s latency and bandwidth demands. This network profiler can also be used to emulate the performance impact of different memory network configurations on the target application without recompilation, facilitating the exploration of cache network cost-performance tradeoffs. Subsequently, LMC analyzes the profiling 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 and passes it through a standard tool flow to produce an FPGA image.

Section 7.1 describes LMC’s extensions to the LEAP compilation flow. With the LMC framework, many resource-aware and application-aware memory optimizations, including both cache microarchitecture optimizations and cache network optimizations, can be automated. In this chapter, we focus on cache network optimizations. We examine some automatic optimizations enabled by LMC in Section 7.2. These optimizations aim to improve program performance by customizing the on-chip cache networks. First, in Section 7.2.1 we present memory partitioning mechanisms that enable user programs to automatically take advantage of the increasing memory capabilities of modern FPGA systems. As FPGAs have grown in size and capacity, FPGA on-board memories have become both richer and more diverse in order to support the increased computational capacity of FPGA fabrics. For example, Nallatech 510T has eight DDR4 memories plus an ultra-fast Hybrid Memory Cube [68], and Intel Xeon+FPGA platform has two DDR3 memory channels plus a cache-coherent QPI interconnect to the Xeon processor [1]. To efficiently utilize board-level memory resources, LMC partitions the user-level private caches into disjoint groups and connects each group to a shared cache implemented on top of each board-level memory. Each partition is connected using a separate network, which serves to increase memory throughput while simultaneously reducing memory latency. To improve memory bandwidth, LMC implements an intelligent memory interleaving mechanism that 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.

To further reduce memory latency, especially for programs with a large number of memory clients, in Section 7.2.3, we develop algorithms to select an optimal topology for each partitioned private cache network. 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). LMC takes the profiling results obtained from the network profiler...
7.1. Compiler Overview

and uses the above optimization techniques to automatically construct an optimized network for the target application.

To test the scalability and robustness of our proposed network construction 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 datacenter context. To support the needs of such deployments and to help amortize large FPGAs, FPGA virtualization has been proposed [1, 28], 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 resources used by competing applications, we introduce some quality-of-service controls into the compiler-generated memory network to control fairness among multiple programs.

In Section 7.3, 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 5% of the total chip area. For hand-assembled, bandwidth-intensive applications, the partitioned ring-based network improves the performance over the baseline memory hierarchy by 42% to 100% (with a 56% geometric mean) because of the increased memory bandwidth. For HLS-compiled applications, which involve pointer-chasing and are latency-sensitive, the network partitioning optimization provides 24% performance improvement on average because of the reduced network latency. Our automatically-generated, program-optimized tree-based networks minimize the impact of network latency and further improve the performance of those applications by 18% (a 45% performance gain over the baseline) on average. For multi-program applications, our bandwidth-controllable tree networks also improves fairness by preventing bandwidth-intensive applications from saturating the memory system bandwidth.

7.1 Compiler Overview

To automate the construction of optimized memory systems tailored to different applications, we extend the LEAP compilation flow [31] by adding a series of compilation
phases, which we refer to as the LEAP Memory Compiler (LMC). LMC operates in three phases: instrumentation, analysis, and synthesis. The combination results in an application-specific memory hierarchy. Figure 7-1 shows the extended compilation flow, which optionally includes profiling compilation where a network profiler is constructed with program instrumentation hardware. The details of the profiler design and how we use it to emulate the performance impact introduced by cache networks will be described in Section 7.2.2. During profiling compilation, the target program with instrumentation hardware injected is executed with one or several test configurations to obtain the runtime information of memory access behavior for each memory client. Then, the instrumentation results are passed to the main compilation where the target program is recompiled to construct an optimized memory hierarchy.

The first phase of LMC is program instrumentation. For many FPGA applications, there are multiple memory clients in the system and the clients may have highly asymmetric behavior 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
7.1. Compiler Overview

Figure 7-2: Program instrumentation built with the LEAP statistics collection service.

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 [95]. However, in order to target more general and more complicated programs including hand-assembled applications, we resort to FPGA-based runtime instrumentation.

Figure 7-2 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 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. To construct program-optimized cache networks, the service connection abstraction introduced in Chapter 6 is used to describe the network endpoints of memory clients and memory controllers that are associated with board-level memories. Similar to the way that the original LEAP compiler connects latency-insensitive modules, LMC gathers memory clients and controllers with the same service name, which are usually the ones mapped to the same FPGA, by parsing the intermediate files produced by the Bluespec compiler. Then, LMC optimizes the cache network by assigning memory clients to available memory controllers and selecting a network topology for each controller group. This phase is optionally feedback-driven: the instrumentation results obtained from previous profiling compilation can be utilized for further optimizations, such as bandwidth-aware partitioning, which we will discuss in Section 7.2.1, and low-latency tree-based networks, which we will discuss in Section 7.2.3. The output of the analysis phase is an abstract representation of the memory hierarchy and 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. Based on the abstract representation of the memory hierarchy obtained from the analysis phase, LMC constructs the physical network connecting memory clients to associated controllers in the top-level Bluespec program for each FPGA platform. After LMC is complete, the original LEAP compilation is resumed to generate the final FPGA implementation through the Bluespec compiler and an FPGA-vendor tool chain.

### 7.2 Construction of Optimized Cache Networks

This section introduces the analysis and mechanisms by which LMC constructs platform-optimized and program-optimized cache networks. We target the cases when there is more than one board-level memory controller on the FPGA and the application has memory clients with asymmetric memory access characteristics. 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.

### 7.2.1 Resource-Aware Program-Optimized Memory Partitioning

To utilize the bandwidth of multiple on-board memories efficiently, we begin by constructing a central cache controller for each board-level memory, as shown in Figure 7-3. The result is a set of distributed caches. Each central cache uses an on-board memory 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 caches 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 networks to connect all the nodes within the same group.
For applications with largely homogeneous memory clients, random partitioning effec-
tively reduces network latency and balances 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 bandwidth-intensive 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 each memory client's traffic, which is the
total number of messages sent from the 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.

Load-balanced partitioning can be classified as a minimum makespan scheduling
problem. The goal is the assign each client to a controller to minimize the makespan,
which is defined as the maximum traffic on any controller network. Suppose we are given
$M$ controllers and $N$ memory clients, and the $i$th client has traffic $t_i$ for each $i = 1, \ldots, N$.
This partitioning problem can also be formulated as an integer linear programming (ILP)
problem. Let $x_{ij}$ be a decision variable that designates whether the $i$th client is assigned to
the $j$th controller for each $i = 1, \ldots, N, j = 1, \ldots, M$. The problem can thus be modeled
as follows:

$$\text{minimize} \quad T$$
$$\text{subject to:} \quad \sum_{i=1}^{N} x_{ij} \cdot t_i \leq T \quad (j = 1, 2, \ldots, M)$$
$$\sum_{j=1}^{M} x_{ij} = 1 \quad (i = 1, 2, \ldots, N)$$
$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, N; j = 1, \ldots, M)$$
Algorithm 7-1 Private cache network partitioning.

1: procedure PARTITION(clientList, controllerList, statsFile)
2: Initialize controller bandwidth to be zero
3: if statsFile exists then
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 bandwidth to c’s bandwidth
10:    Remove s from clientList
11:   end while
12: else
13:   n ← Length of controllerList
14:   sLists ← Randomly separate clientList into n lists
15:   for i in 1 to n do
16:     Connect sLists[i] to controllerList[i]
17:   end for
18: end procedure

LMC adapts the classical longest-processing-time (LPT) algorithm [41], a polynomial-time approximation scheme, to solve the memory bandwidth partitioning problem. This algorithm approximates optimal load balancing by assigning new memory traffic to the least-loaded memory controller. Algorithm 7-1 describes how LMC partitions the private cache network on a single FPGA. 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 above 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 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. With the memory interleaver, the integrality constraints of the original ILP formulation can be removed and the partitioning problem is relaxed to a linear program. Since the
constraints are relaxed, the partitioning result is very likely to be improved. However, the memory interleaver logic also introduces extra latency. Therefore, the latency-bandwidth tradeoff needs to be considered when applying this interleaving technique.

When the original partitioning approach produces highly unbalanced partitions, for example, when there is only one memory client in the system or when some of the clients has much higher bandwidth demand, 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 original partitioning method described in Algorithm 7-1: LMC first deals with the memory clients whose memory needs to be interleaved, assigning them to multiple controllers, and then apportions the remaining non-interleaved memory clients.

Figure 7-3 illustrates an example of a partitioned cache network with two on-board DRAMs in the system. In this example, there is one private memory that connects to two controller networks via the memory interleaver logic. The microarchitecture of the private memory interleaver logic is shown in Figure 7-4. When the memory interleaver receives a request from the associated private memory, it forwards the request to one of the controller networks based on the target request's word-level address. We route
7.2. Construction of Optimized Cache Networks

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.

The partitioning mechanisms introduced in this section are designed for private memory networks. For applications using shared memory, the coherent cache network optimization, which partitions the shared memory address space into multiple regions and connects the memory components through hierarchical rings as described in Section 4.2, can be applied to better utilize the board-level memory resources. Moreover, since the coherent memory hierarchy is integrated with the private memory hierarchy, the private cache network optimizations can be directly composed with the coherent cache network optimization to further improve overall system performance.

7.2.2 Cache Network Profiler

Through cache network partitioning, programs can effectively utilize increased memory bandwidth from modern FPGA boards. For programs that consist of a large number of memory clients, in order to further reduce the performance impact introduced by network latency, it is important to select a network topology that achieves better scalability. Topology selection requires the examination of network design tradeoffs for each 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.

To emulate different networks without recompilation, our network profiler utilizes the LEAP dynamic parameter service, which allows parameter values to be overwritten at runtime through command-line switches. Figure 7-5 illustrates how a LEAP program instantiates dynamic parameters whose values can be overwritten at runtime. A dynamic parameter can be instantiated by specifying a unique name and a default value. A module that needs to access any dynamic parameters requires a parameter node to receive updated values. At compile time, the compiler connects all parameter nodes with a centralized controller via a latency-insensitive ring. In addition, the compiler gathers all dynamic parameters and records them in a table on the host processor. During execution, the software on the host processor updates the parameters with values received from the command line and sends the updated values to the dynamic parameter service controller on the FPGA. The controller then broadcasts the values of all the parameters. Each parameter node checks the broadcast messages on the ring and sends the values of the associated parameters to the module.

Figure 7-6 shows an example of an application-specific network profiler, which is automatically constructed during profiling compilation.
7.2. Construction of Optimized Cache Networks

Figure 7-6: An application-specific network profiler with instrumentation logic and latency FIFOs inserted at each memory client.

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.

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 the memory interleaver logic shown in Figure 7-4. In the memory interleaver logic, the size of each interleaved memory region is also dynamically configurable. Each memory client can thus 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 controller through N-to-1 tree routers, where
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 6.3, is also dynamically configurable, enabling the evaluation of different bandwidth allocation strategies.

The latency FIFOs, tree routers, and memory interleavers in the network profiler are all dynamically configurable. This enables the profiler to emulate the performance of different network topologies, such as a single 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 we insert a DRAM performance controller that matches the DRAM latency and throughput to the profiled network. In addition, although the bandwidth allocation for each client is controllable, it may still be difficult to perfectly model network contention, resulting in lower emulation accuracy for programs that have bandwidth-hungry clients.

### 7.2.3 Program-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
7.2. Construction of Optimized Cache Networks

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 algorithms introduced in Section 7.2.1, 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 are many leaf nodes. 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 $n$th 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 sum of the weight values for all the leaf nodes in a tree 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. Without loss of generality, we assume all non-leaf nodes (which are the tree routers including the root) 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 weight value of the tree 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 leaf nodes 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 $n$th leaf node at depth $d$ for each $n = 1, \ldots, N, d = 1, \ldots, D$. We can formulate the topology synthesis problem as an ILP problem with the following decision variables for each $n = 1, \ldots, N, d = 1, \ldots, D$:

$$\lambda_{nd} \in \{0, 1\} : \text{whether the } n\text{th leaf is at depth } d$$

$$x_d \in \mathbb{Z}_{\geq 0} : \text{number of leaf nodes at depth } d$$

$$y_d \in \mathbb{Z}_{\geq 0} : \text{number of non-leaf nodes at depth } d$$

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

$$\text{minimize } \sum_{n=1}^{N} \sum_{d=1}^{D} w_{nd} \cdot \lambda_{nd}$$

subject to:

$$\sum_{d=1}^{D} \lambda_{nd} = 1 \quad (n = 1, 2, \ldots, N)$$

$$x_d = \sum_{n=1}^{N} \lambda_{nd} \quad (d = 1, 2, \ldots, D)$$

$$y_0 = 1, \quad y_d + x_d = K \cdot y_{d-1} \quad (d = 1, \ldots, D)$$

$$\lambda_{nd} \in \{0, 1\} \quad (n = 1, \ldots, N; d = 1, \ldots, D)$$

$$x_d \in \mathbb{Z}_{\geq 0}, \quad y_d \in \mathbb{Z}_{\geq 0} \quad (d = 1, \ldots, D)$$

For each leaf node $n$, if its weight values can be approximated as an affine function of depth $d$: $w_{nd} = a_n \cdot d + c_n, a_n \geq 0$, the tree topology synthesis problem can be solved using dynamic programming (DP), reducing the problem complexity to polynomial
Algorithm 7.2 Construct a minimum weight tree using DP.

1: procedure TREECONSTRUCTION(N, D, K, \{a_n\})
2: Sort \{a_n\} so that \(a_1 \leq a_2 \leq \cdots \leq a_N\)
3: \(V \leftarrow \text{inf}\) \(\triangleright V[d][b][m]: \text{costs}\)
4: for \(d = D, D - 1, \ldots, 1\) do \(\triangleright \text{Base case: } d = D\)
5: for \(b = K, 2K, \ldots, \min\left(\left\lceil \frac{N}{K} \right\rceil, (K - 1)^d \cdot K\right)\) do
6: for \(m = 1, 2, \ldots, N\) do
7: if \(b > m\) then \(\triangleright \text{Place all } m \text{ leaf nodes at depth } d\)
8: \(V[d][b][m] \leftarrow d \cdot \sum_{n=1}^{m} a_n\)
9: else if \(d \neq D\) then
10: \(V[d][b][m] \leftarrow \text{FINDMIN}(V, K, \{a_n\}, d, b, m)\)
11: end if
12: end for
13: end for
14: end procedure
15: function FINDMIN(V, K, \{a_n\}, d, b, m)
16: return \(\min_{x=0,1,\ldots,b-1} \left( d \cdot \sum_{n=m-x+1}^{m} a_n + V[d+1][((b-x)\cdot K)[m-x]] \right) \)
17: end function

der time. We first sort and re-index the leaf nodes so that \(a_1 \leq a_2 \leq \cdots \leq a_N\). 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 7.2, 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, bandwidth upper limits are set for leaf nodes that have large bandwidth demands, in order to control fairness and prevent bandwidth-intensive applications from saturating DRAM bandwidth and slowing down latency-sensitive applications.

7.3 Evaluation

The majority of our evaluation targets the Xilinx VC709 platform, which includes two physical memory controllers, each connected to 4 GB of DRAM. We use these DRAMs to implement two board-level caches for each VC709 FPGA. We also test our memory partitioning techniques on two networked FPGA deployments: a dual VC707 and a dual VC709 configuration.

To evaluate the benefit of LMC, we examine a set of benchmarks with different memory access patterns: memperf, heat, mergesort, filter, and reflect-tree, as described in Section 2.3. Heat operates on a parametric data size. In this section, we examine heat with 16 worker engines and two data size parameterizations: 8-bit, which gives a degree of spatial locality in the coherent cache, and 64-bit, which has less locality. Mergesort itself has a parametric memory system and can instantiate several parallel, banked memory interfaces to improve memory bandwidth. We examine two configurations: a baseline configuration with a single private memory per sorter and a configuration with two memory interfaces, which we refer to as banked. The implementation of both filter and reflect-tree is split into 8 independent partitions ($P = 8$), resulting in 24 and 16 LEAP private memories.

We utilize Vivado HLS to transform the HLS benchmarks (filter and reflect-tree) into RTL implementations and employ Xilinx Vivado 2015.1 for all synthesis and physical implementation work. Also, we use the Gurobi optimizer [42] to solve the ILP problems. All resource utilization and clock rate results reported in this section are post-place-and-route results.

To show the performance benefit of program-optimized cache networks, we compare the performance of our compiler-generated networks with a baseline LEAP memory
7.3. Evaluation

hierarchy. This memory hierarchy has a single controller hierarchy per FPGA to manage accesses to all available DRAMs on the FPGA, and all memory clients on the same FPGA are connected with the controller via global LEAP rings. We refer to this implementation as baseline in this section.

7.3.1 Cache Network Partitioning

We start with the evaluation of our program-optimized memory partitioning mechanisms enabled by LMC. To emphasize the benefit of partitioning, the components in each partitioned network are connected via a single ring as depicted in Figure 6-3a. More complicated network topologies will be evaluated in Section 7.3.2. In addition, all tests in this subsection run at 100 MHz to ensure performance results are comparable.

7.3.1.1 Basic Memory Behavior

To build intuition about the behavior of our optimized memory systems, we benchmark three memory system implementations – a baseline implementation (baseline) and two different interleaved implementations – using memperf, a memory performance kernel that varies both the working set size and reference locality. As shown in Figure 7-7, for each configuration, performance is divided into two regions. In regions of high locality, the throughput is up to one word per cycle. 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. In the low-locality regions, when all accesses are serviced in the central caches, our interleaving techniques nearly double the throughput over the baseline 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,

\[1\] There is a small region of increased bandwidth in the baseline memory system because of the FPGA-side on-chip caching of DRAM lines.
reverting to baseline performance. To combat this case, we introduce address hashing which recovers most of the DRAM bandwidth by evenly balancing the interleaving.

7.3.1.2 Area Consumption

Table 7-1 describes the area requirements of various components of the LEAP memory hierarchy. We consider two central cache controller implementations: the unified cache controller, which is constructed in the baseline memory system to service two DRAMs, and the single cache controller, which services a single DRAM. Managing a second on-board DRAM marginally increases the area utilization of the cache controller, due to the increasing width of buses within the controller.
Table 7-1: FPGA resource utilization for memory system components.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>18K-bit BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Network Ring Stop</td>
<td>680</td>
<td>626</td>
<td>0</td>
</tr>
<tr>
<td>Memory Interleaver</td>
<td>1575</td>
<td>1766</td>
<td>0</td>
</tr>
<tr>
<td>Unified Central Cache Controller</td>
<td>14,499</td>
<td>16,513</td>
<td>18</td>
</tr>
<tr>
<td>Single Central Cache Controller</td>
<td>13,195</td>
<td>15,376</td>
<td>18</td>
</tr>
<tr>
<td>DRAM Controller</td>
<td>8525</td>
<td>13,661</td>
<td>0</td>
</tr>
<tr>
<td>Private Memory Client</td>
<td>1660</td>
<td>2010</td>
<td>4</td>
</tr>
<tr>
<td>Coherent Memory Client</td>
<td>2985</td>
<td>5721</td>
<td>7</td>
</tr>
<tr>
<td>Coherence Controller</td>
<td>6795</td>
<td>7658</td>
<td>19</td>
</tr>
</tbody>
</table>

Within the memory system, the main 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 main 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 is a small price to pay for the performance gains we will describe in subsequent sections.

The main overhead of our intelligent network synthesis is the introduction into the memory network of new ring stops. In the case where we simply assign memory clients to different memory networks without interleaving, no new hardware is introduced. Memory network ring stops that are capable of address interleaving, shown in Figure 7-4, are more than twice the cost of baseline ring stops, since interleaved ring stops must communicate with two or more controller networks. However, the cost of ring stops, 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 7-2 shows the area utilization of baseline and best-performing implementations optimized by our memory partitioning mechanisms for each of our single-program applications. As expected, LMC partitioning 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
Table 7-2: Resource utilization for baseline and best performing memory configurations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>18K-bit BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memprof</td>
<td>Baseline</td>
<td>62,317</td>
<td>80,722</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>84,166</td>
<td>105,308</td>
</tr>
<tr>
<td>Mergesort</td>
<td>Baseline</td>
<td>99,284</td>
<td>137,314</td>
</tr>
<tr>
<td>(4 Sorters)</td>
<td>Optimized</td>
<td>110,116</td>
<td>146,592</td>
</tr>
<tr>
<td>Heat</td>
<td>Baseline</td>
<td>157,361</td>
<td>229,276</td>
</tr>
<tr>
<td>(8-bit Data)</td>
<td>Optimized</td>
<td>189,684</td>
<td>264,881</td>
</tr>
<tr>
<td>Filter</td>
<td>Baseline</td>
<td>158,668</td>
<td>192,784</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>177,154</td>
<td>209,215</td>
</tr>
<tr>
<td>Reflect Tree</td>
<td>Baseline</td>
<td>118,692</td>
<td>141,210</td>
</tr>
<tr>
<td></td>
<td>Optimized</td>
<td>137,357</td>
<td>159,047</td>
</tr>
<tr>
<td>Average Utilization Increase</td>
<td>19,244</td>
<td>18,851</td>
<td>25</td>
</tr>
<tr>
<td>VC709 Area (%)</td>
<td>2.22%</td>
<td>4.35%</td>
<td>0.85%</td>
</tr>
</tbody>
</table>

in utilization is small relative to the size of the VC709: LUT utilization increases by 4.35% of the full VC709 while register utilization increases by 2.22%.

### 7.3.1.3 Randomized Partitioning

As a baseline for LMC partitioning 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. Figure 7-8, 7-9 and 7-10 show the relative performance of four benchmarks under randomized allocation. In Figure 7-8, the columns represent the average sorter performance normalized to the baseline implementation, and the error bars are used to depict the minimum and maximum sorter performance instead of the standard deviation. In Figure 7-9, the performance results are normalized to the baseline implementations in which both private and coherent cache network optimizations are disabled. In general, random partitioning is successful in improving program performance, especially for symmetric applications like mergesort. On the other hand, although coherent memories in heat are largely symmetric in their runtime behavior, heat's coherence controllers include private memory clients for data and ownership, and these are asymmetric: the memory client for data storage uses ten times the bandwidth of the client for ownership.
7.3. Evaluation

Figure 7-8: Performance of *mergesort* on 256 kilo-entry lists with different memory network configurations and a various number of parallel sorters, averaged over all the sorters and normalized to the baseline implementations.

Figure 7-9: Performance of *heat* on a 1-mega-entry array with various memory network configurations, normalized to the baseline implementations with a single coherence controller.

storage. For bandwidth-intensive applications with asymmetric memory clients like *heat*, random partitioning gives only limited performance gains.

Like *heat*, the HLS applications (*filter* and *reflect-tree*) also features asymmetric memory clients. However, random allocation actually performs as well as (or even slightly better than) more sophisticated allocation schemes. This is because these HLS applications chase data-dependent pointers and thus have limited ability to produce multiple outstanding
memory requests per memory client. Therefore, they are less sensitive to bandwidth balancing but extremely sensitive to latency. Random partitioning both halves and balances memory network latency relative to the baseline, and, as a result, improves the performance by 22% for filter and 27% for reflect-tree.

7.3.1.4 Load-balanced Partitioning

The chief weakness of random partitioning is that it can sometimes oversubscribe the bandwidth of a single memory client, 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. For example, when the coherent cache network optimization introduced in Section 4.2 is applied to the heat implementation to construct dual coherence controllers, resulting in four asymmetric private memories, load-balanced partitioning outperforms random partitioning. As shown in Figure 7-9, compared to random partitioning, heat obtains another 7%-43% performance gain with load balancing, since the heavily-loaded clients, which are the private memories used for coherence data storage, are evenly spread across the two board-level memories of the VC709.
7.3. Evaluation

Load balancing naturally preserves the performance of symmetric applications like mergesort. For applications that are more sensitive to memory latency instead of bandwidth, load-balanced partitioning sometimes provides less performance gains compared to random partitioning. This is because load balancing evens out bandwidth, but ignores latency: low-bandwidth memory clients may all be assigned to the same network. For example, in the case of reflect-tree, the memory network latencies are slightly imbalanced due to bandwidth balancing, which results in a small performance degradation compared to random partitioning.

7.3.1.5 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 7-8, a three-sorter instantiation of mergesort experiences some benefit under load-balancing, but achieves less of a performance gain than either the two- or four-sorter case, especially for the sorter engine with minimum performance. This is because two of the sorters in the three-sorter case must share a single controller. If we introduce memory interleaving, the memory accesses can be spread across controllers equally, leading to further performance gains (especially for the worst-performing sorter). Similarly, applications like memperf or a single-sorter mergesort, which have only one memory client, can benefit from multiple controllers when using our interleaving approach.

The banked versions of mergesort, which try to improve memory bandwidth at user level by allocating two memory clients per sorter, achieve performance gains similar to those of interleaving when the partitioning technique is applied. However, if without partitioning, the banked versions add latency and promulgate queuing delay in the memory network, lowering performance for large numbers of sorters.

Bandwidth-intensive applications generally benefit, or at least maintain load-balanced performance, with address interleaving. However, for latency-sensitive applications, like
filter and reflect-tree, the extra cycles of latency introduced by the interleaver and additional ring stops result in a performance degradation. This suggests that care must be taken by the compiler when applying network optimizations to latency-sensitive applications.

7.3.1.6 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 introduced in Section 4.2. Figure 7-9 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 25% in the best, load-balanced configuration.

Memory interleaving of the coherence domain at the coherence controller level provides a performance gain around 10%-30%. 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 100% for the 64-bit version of the heat benchmark and a 55% 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.

7.3.1.7 Multiple FPGAs

Looking forward to cloud deployments [78] 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.
7.3. Evaluation

Figure 7-11: Performance of best-achieved performance solutions for 8-bit heat and mergesort with various platform configurations, normalized to single-VC707 solutions.

located on a nearby FPGA by automatically constructing a network between the FPGAs. The results of this experiment are shown in Figure 7-11.

Since inter-FPGA networks add latency, we examine only the bandwidth-intensive benchmarks: heat and mergesort. 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. Mergesort 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 remote memory. We note that it is unrealistic to expect designers to consider such complex alternative systems without the assistance of a compiler to manage communication and resource allocation.

Scaling mergesort across two VC709 FPGAs and to four board-level memories yields another significant performance improvement. This is particularly pronounced for the eight sorter case because the baseline, which is the single private memory controller case, has a large single-ring 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

---

1Heat results are normalized to the performance of the implementation 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. Mergesort results are normalized to the optimized single-VC707 solution.
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 bandwidth-intensive applications, that bandwidth borrowing or sharing among adjacent, networked FPGAs can be a significant source of performance gains.

7.3.2 Program-Optimized Cache Networks

As discussed in Section 7.3.1, memory partitioning achieves smaller performance gains on filter and reflect-tree compared to the other benchmarks we study, because these two applications are more sensitive to memory latency instead of bandwidth. To improve the performance further, it is essential to construct more complicated cache networks that minimize the network latency impact.

To demonstrate the performance benefit of low-latency networks, we compare the performance of our compiler-optimized networks with two existing memory network configurations: baseline and a partitioned ring network constructed using our load-balanced partitioning mechanism, which is the balanced partition configuration shown in Figure 7-10. We refer to the latter as the single-ring network configuration in this subsection. 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.

In addition to filter and reflect-tree, which are single-program HLS applications, we also set up the following multi-program applications to evaluate our bandwidth-controllable tree networks:

**Mergesort-Filter:** A multi-program application in which mergesort with 4 sorters is constructed and scheduled to run with 8-partition filter (P = 8) at the same time. This
7.3. Evaluation

Table 7-3: Resource utilization for various network primitives.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>P&amp;R f_max (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring Node/Connector</td>
<td>680</td>
<td>626</td>
<td>400</td>
</tr>
<tr>
<td>(K = 3)</td>
<td>577</td>
<td>437</td>
<td>179</td>
</tr>
<tr>
<td>(K = 5)</td>
<td>811</td>
<td>826</td>
<td>143</td>
</tr>
<tr>
<td>Tree Router (K = 8)</td>
<td>951</td>
<td>1337</td>
<td>132</td>
</tr>
<tr>
<td>(K = 16)</td>
<td>2840</td>
<td>4501</td>
<td>91</td>
</tr>
<tr>
<td>(K = 32)</td>
<td>11,866</td>
<td>19,390</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 7-4: Resource utilization for the cache network in filter.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Slice Registers</th>
<th>Slice LUTS</th>
<th>P&amp;R f_max (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Ring</td>
<td>22,968</td>
<td>17,321</td>
<td>400</td>
</tr>
<tr>
<td>Hierarchical Ring</td>
<td>24,811</td>
<td>18,187</td>
<td>400</td>
</tr>
<tr>
<td>Tree ((K = 3, ILP))</td>
<td>12,914</td>
<td>9992</td>
<td>179</td>
</tr>
<tr>
<td>Tree ((K = 6, ILP))</td>
<td>10,760</td>
<td>7990</td>
<td>139</td>
</tr>
</tbody>
</table>

multi-program application has 28 LEAP private memories in total: 4 from mergesort and 24 from filter.

**Heat-Filter**: A multi-program application that includes heat with 8 worker engines and 4-partition filter \((P = 4)\). The shared memory space for heat in this multi-program application is interleaved and is managed by dual coherence controllers. The resulting implementation has 16 LEAP private memories: 4 from heat and 12 from filter.

7.3.2.1 Resource Utilization

Table 7-3 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 7-4 shows the resource 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 7.3.2.2, tree-based networks with a smaller $K$ are able to maintain the target frequency and provide better performance compared to ring-based networks.

7.3.2.2 Single-Program Applications

To construct a tree-based network that minimizes the network latency impact on program performance given an upper bound of $K$, LMC first measures the latency sensitivity of each memory client using the network profiler. Figure 7-12 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 7.2.3. For each memory client in filter, its

Figure 7-12: Latency sensitivity of memory clients in filter.
weight values form a straight line and therefore can be approximated as an affine function of tree depth, allowing LMC 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 7.2.3. The memory clients that store stack data structures are not shown in Figure 7-12 because they have 100% hit rate in the first-level caches and therefore have zero latency sensitivity. The memory clients that store 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. \textit{Balanced-tree (random)} first constructs a balanced tree and then randomly assigns clients to leaf nodes. This approach is also used by LMC when the profiling compilation is disabled and latency sensitivity of each client is unknown. \textit{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 leaf nodes that have a smaller tree depth.

Figure 7-13: Simulated performance of filter with various tree construction algorithms.
Figure 7-13 shows the simulated performance of filter built with the four tree construction algorithms at a varying tree radix $K$. The performance speedup is calculated by comparing the runtime cycles measured from the network profiler to the runtime of the actual baseline implementation. Since the 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 7-13, 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 $K = \text{Max}$). 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 7-14 shows both the simulated performance and the actual performance of various network configurations for single-program applications. For each application, the performance speedup is calculated by comparing the runtime cycles measured from the profiler and from the actual, physical implementation to the runtime cycles.
of the actual baseline implementation. 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 outperforms the baseline by 47% and the single-ring configuration by 16%. 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 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.

7.3.2.3 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 \( \text{Performance}_{MP} \) is the program performance when executing with other programs and \( \text{Performance}_{SP} \) is the performance measured when executing alone. We also adopt the following fairness metric proposed in [92]:

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

where \( n \) is the number of programs and \( r_i \) is the performance ratio of the \( i \)th program. This fairness metric, which ranges from zero to one, is a harmonic mean of the performance ratios.

Figure 7-15 shows the performance comparison of various network configurations when filter and mergesort are scheduled to run simultaneously. To make a fair comparison,
Chapter 7. LEAP Memory Compiler

we control the number of iterations filter executes so that filter and mergesort 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. Figure 7-15a and 7-15b show the performance of the two programs under different program configuration settings, where C indicates the program hardware is constructed on the FPGA and E indicates the program is executed. For each program, the performance result is normalized to that of the implementation where the program is constructed alone with the baseline memory network. As shown in Figure 7-15a, filter performance slows down a little for baseline and single-ring configurations when mergesort is constructed on the FPGA due to the increased network latency introduced by four additional clients. When filter is executing with mergesort, if without bandwidth control, the performance slowdown is over 50% because mergesort saturates the memory

Figure 7-15: Performance comparison of various network configurations for the mergesort-filter application.
bandwidth, while our bandwidth-controllable tree network reduces the performance slowdown to 5% by limiting the bandwidth consumption of mergesort clients.

As shown in Figure 7-15b, when filter is constructed, the mergesort 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, mergesort performance does not slow down because filter only consumes little memory bandwidth. Adding bandwidth limitation to mergesort clients degrades the mergesort performance by 13% but achieves much better fairness as shown in Figure 7-15c.

We also evaluate the performance ratios and fairness for heat-filter with different network configurations, as shown in Figure 7-16. Similar to mergesort, 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 mergesort 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.

7.4 Summary

Given the memory and communication abstractions as well as the configurable memory building blocks introduced in previous chapters, the final step for automating the construction of program-optimized memory hierarchies is to develop a build procedure. This build procedure should be able to automatically select and configure the available building blocks based on the memory access characteristics of the target application as well as the available resources on the target platform. In this chapter, we have presented the LEAP Memory Compiler (LMC), which can automate resource-aware, application-specific optimizations through feedback-directed compilation. We have demonstrated that LMC can transparently optimize the memory system of a given application by examining two memory network optimizations enabled by LMC.

As FPGAs and FPGA applications have scaled, modern FPGA boards usually include multiple board-level memories to support the increased bandwidth demand of the user program. In addition, as on-chip memory systems have also scaled to support FPGA applications with a large number of processing engines, memory network latency has sometimes become a serious design consideration. The memory network optimizations introduced in this chapter are designed to automatically construct program-optimized cache networks in order to alleviate the complexity of designing high-performance applications for such systems. Because of the memory and communication abstractions, LMC can freely perform these memory network optimizations without changing the rest of the system. Since constructing a program-optimized cache network requires the exploration of a large design space for each application, to facilitate such exploration process, we propose a dynamically-configurable network profiler that can evaluate the performance impact introduced by different topologies of the cache network without recompilation. This network profiler can also be used to characterize the latency and bandwidth requirements of each memory client in the target application. Using the profiling statistics, LMC
7.4. Summary

automatically partitions the network that connects user-specified memory interfaces to board-level memories, simultaneously increasing the memory bandwidth and reducing memory latency. A load-balanced partitioning mechanism is adopted to balance the total traffic on each of several partitioned memory networks so that asymmetric memory clients can efficiently share the memory bandwidth. In addition, we seek to further alleviate the latency issue present in scaled out FPGA memory systems through the construction of program-optimized tree-based networks. Our evaluation, which targets platforms with multiple board-level memories, shows that our compiler-optimized cache networks provide significant performance gains.

Although we have focused on automatic cache network optimizations, the proposed LMC framework can also be used to automate other cache optimizations, such as those proposed in Chapter 5. One direction of future work is to explore those optimizations.
FPGAs have become increasingly popular as accelerators because of their energy-efficiency and performance characteristics. However, as FPGAs and FPGA-based systems have grown, traditional approaches such as low-level hardware development and system-level hand-tuning have become too complicated. To address FPGA programmability challenges, several FPGA high-level abstractions have been proposed, but they sometimes come at the expense of performance. In this thesis, we have demonstrated that, with automatic, resource-aware, application-specific optimizations performed under these abstractions, it is feasible to reduce the performance gap between a generated system and a manually-optimized design. This chapter summarizes the contributions of this thesis and outlines future research directions.

8.1 Thesis Summary

This thesis primarily focuses on optimizing FPGA-based memory systems. Unlike general-purpose processors where the memory hierarchy is usually fixed at design time based on a set of expected workloads, FPGA memory systems can be tailored to different applications at compilation time based on the properties of those specific applications. This thesis achieves the goal of automating the construction of program-optimized memory hierarchies through three steps: (i) defining memory abstractions to decouple the functionality of memory systems from physical implementations, (ii) providing optimizations that can be performed under these abstractions to construct memory building blocks
with different design tradeoffs, (iii) developing a build procedure to automatically select and implement an optimized memory hierarchy tailored for the target application and the target platform based on the program introspection feedback. We show the effectiveness of the proposed optimizations by examining a set of benchmarks, ranging from small kernels to large programs, with different memory access characteristics and targeting several FPGA platforms.

To optimize the memory system without changing the user program, this thesis adopts the LEAP private memory [3] as the basic memory abstraction and extends it to support a wider variety of applications. In Section 2.2, we extend the LEAP memory interface to support software application kernels compiled through HLS and design wrappers to embed the HLS kernels into the LEAP platform. These HLS kernels access LEAP memories through standard bus protocols. In Chapter 4, we extend the LEAP memory support to parallel algorithms written with a shared-memory programming model by introducing a shared memory abstraction to maintain coherency and consistency of shared-memory accesses for programmers. This shared memory abstraction consists of LEAP coherent memories, which manage coherent caches on a single FPGA or across multiple FPGAs while presenting a simple, abstract user interface, as well as FPGA-based synchronization primitives, including locks and barriers. In addition to memory abstractions, we adopt a similar idea and introduce a new communication abstraction in Chapter 6 to enable a systematic exploration of different cache network configurations, which is important for accelerating applications with multiple asymmetric, but simultaneously active memory clients.

To facilitate the construction of program-optimized memory hierarchies, in Chapter 5, we examine several configurable cache optimizations performed under the LEAP memory abstractions, including cache prefetchers and scalable on-chip caches. These optimizations exploit spare resources to speed up programs and can be configured to construct a variety of memory building blocks from which an optimized memory hierarchy can be formed. Experimental results show that adding prefetchers to a synthesized memory hierarchy improves performance of pre-existing programs by 17% on average. We also have good success in building large first-level caches. Across our benchmark suite, our scalable
8.1. Thesis Summary

cache microarchitecture achieves performance gains of 7% to 74% (with a 26% geometric mean) over the baseline cache microarchitecture. Another contribution is to provide beneficial directions in the exploration of the memory hierarchy space and to develop a framework that assists such exploration by allowing programmers and compilers to configure a multi-level memory hierarchy, to enable or disable certain cache optimization through parameters.

A main contribution of this thesis is in the automated optimization of memory systems. Armed with memory abstractions and configurable components in the memory hierarchy, in Chapter 7, we present the LEAP Memory Compiler (LMC), an extension to the original LEAP compilation flow [31]. LMC is a feedback-driven compiler that automatically customizes an optimized memory topology connecting user-level memory interfaces to board-level memories based on available memory resources and program runtime statistics collected during profiling compilation. To characterize latency and bandwidth demands of memory clients in the target application, a dynamically-configurable network profiler with program introspection logic inserted is proposed. This profiler is synthesizable on FPGA and can also be used to emulate different cache network topologies without recompilation.

LMC incorporates several application-specific optimizations that take advantage of both interface abstraction and the availability of extra memory resources. These optimizations include memory partitioning techniques, which allow programs to automatically make use of increased memory bandwidth on modern FPGAs, and the construction of optimized tree-based networks that minimize the performance impact introduced by network latency. Our experiments show that, for throughout-oriented applications running on a single FPGA with two board-level memories, the proposed memory partitioning mechanisms efficiently utilize the increased board-level memory bandwidth and achieves 56% performance improvement on average over the baseline memory hierarchy. The low-latency, tree-based cache network further improves the performance of latency-oriented applications, which benefit less from memory partitioning, and achieves a performance gain of 45% over the baseline.
8.2 Future Work

The LMC framework presented in this thesis enables the automation of resource-aware, application-specific optimizations that can be guided by runtime program introspection. We have examined several cache network optimizations, but there are many other optimizations that can be automated by LMC. This thesis directly builds upon the LEAP FPGA operating system [35], extending its compilation flow and memory system support to improve both the usability and efficiency of LEAP memories. There are other extensions and optimizations can also be applied to achieve further improvement. This section provides some of these extensions as future research directions.

**Automatic cache optimizations:** In Chapter 5, we introduce several cache optimizations, including prefetching and scalable on-chip caches, and provide a framework to configure, enable, or disable these optimizations through parameters. However, these parameters currently need to be manually chosen by application designers, although we have provided some general suggestions. Automatically picking effective optimizations and distributing unused on-chip resources across a memory hierarchy to achieve optimal performance likely require sophisticated program analysis. Such analysis can be incorporated into the LMC framework. For example, we can adopt utility-based cache partitioning (UCP) [79] to automatically determine the first-level cache size for each memory client. UCP is frequently used in processor architectures to partition a shared cache among multiple applications at runtime based on the utility information of each application, which indicates how much the application is likely to benefit from the cache capacity increase. This information is measured via a utility monitor inserted at each core, estimating the reduction in cache misses for each application when a certain amount of cache resources is allocated. Taking advantage of the FPGA configurability, instead of partitioning a shared cache at runtime, the on-chip cache resources can be distributed statically among memory clients. For a given application, utility monitors can be instantiated at each memory client during profiling compilation and the utility estimation of each client can be passed to the main compilation to determine the cache sizes. Similarly, by extending LMC with more
8.2. Future Work

detailed program introspection and analysis, we also expect to be able to make decisions about cache associativity and cache replacement policies.

Optimizations for asymmetric memory controllers: In this thesis, 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 [1, 68], one direction for future work is to explore resource-aware memory network optimizations for asymmetric memory controllers with different latency and bandwidth characteristics.

Energy-related optimizations: The cache and network optimizations introduced in this thesis aim to improve program performance by speeding up the execution time. In Section 5.2.6.5, we show that performance-optimized cache hierarchies may not always achieve the best energy efficiency. An avenue for future research is to investigate our caching scheme in the context of power consumption and to consider energy-performance tradeoffs when deciding program-optimized cache hierarchies.

Partial recompilation: The LMC framework currently consists of two full FPGA compilations: the profiling compilation and main compilation. Because of layers of abstractions, only a part of the memory system, including the program introspection logic, the cache network, and potentially the first-level caches, may be changed between the two compilations while the rest of the system remains to same. We plan to isolate that part of the memory system and investigate the possibility of partial recompilation, shortening the runtime of our feedback-driven compilation.

More applications: We also plan to examine more applications that can be used to demonstrate the benefits of LMC. In particular, we intend to explore applications in cloud deployments. Such applications are likely to have a large number of simultaneously active memory clients with different latency and bandwidth demands and therefore can possibly benefit more from customized optimizations.
A completely automated flow for HLS-kernel integration: With the integration flow introduced in Section 2.2, only the HLS applications that use a single, multi-ported LEAP private memory can be automatically integrated. To further improve the usability of LEAP memory services, we plan to extend the automatic integration flow to support HLS applications that use multiple LEAP private memories for disjoint address spaces as well as applications that need coherency and synchronization support.


Bibliography


Bibliography


