(FPL 2015) Scavenger: Automating the Construction of Application-Optimized Memory Hierarchies

HSIN-JUNG YANG, CSAIL, Massachusetts Institute of Technology
KERMIN FLEMING, SSG, Intel Corporation
FELIX WINTERSTEIN, CAS, Imperial College London
MICHAEL ADLER, SSG, Intel Corporation
JOEL EMER, CSAIL, Massachusetts Institute of Technology

High-level abstractions separate algorithm design from platform implementation, allowing programmers to focus on algorithms while building complex systems. This separation also provides system programmers and compilers an opportunity to optimize platform services on an application-by-application basis. In field-programmable gate arrays (FPGAs), platform-level malleability extends to the memory system: Unlike general-purpose processors, in which memory hardware is fixed at design time, the capacity, associativity, and topology of FPGA memory systems may all be tuned to improve application performance. Since application kernels may only explicitly use few memory resources, substantial memory capacity may be available to the platform for use on behalf of the user program. In this work, we present Scavenger, which utilizes spare resources to construct program-optimized memories, and we also perform an initial exploration of methods for automating the construction of these application-specific memory hierarchies. Although exploiting spare resources can be beneficial, naïvely consuming all memory resources may cause frequency degradation. To relieve timing pressure in large block RAM (BRAM) structures, we provide microarchitectural techniques to trade memory latency for design frequency. We demonstrate, by examining a set of benchmarks, that our scalable cache microarchitecture achieves performance gains of 7% to 74% (with a 26% geometric mean on average) over the baseline cache microarchitecture when scaling the size of first-level caches to the maximum.

CCS Concepts: • **Computer systems organization → Reconfigurable computing;**

Additional Key Words and Phrases: FPGA, memory hierarchy, resource-aware optimization, scalable cache

ACM Reference Format:
Hsin-Jung Yang, Kermin Fleming, Felix Winterstein, Michael Adler, and Joel Emer. 2017. (FPL 2015) scavenger: Automating the construction of application-optimized memory hierarchies. ACM Trans. Reconfigurable Technol. Syst. 10, 2, Article 13 (March 2017), 23 pages.
DOI: http://dx.doi.org/10.1145/3009971

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) have great potential to improve the performance and power efficiency of applications that traditionally run on general-purpose processors. However, the difficulty of designing hardware at register transfer level (RTL) has inhibited the wide adoption of FPGA-based solutions. To reduce

Authors' addresses: H.-J. Yang and J. Emer, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA; emails: {hjyang, emer}@csail.mit.edu; K. Fleming and M. Adler, Intel Corporation, 75 Reed Road, Hudson, MA 01749, USA; emails: {kermin.fleming, michael.adler}@intel.com; F. Winterstein, Department of Electrical and Electronic Engineering, Imperial College London, South Kensington Campus, London SW7 2BT, United Kingdom; email: f.winterstein12@imperial.ac.uk.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 1936-7406/2017/03-ART13 $15.00
DOI: http://dx.doi.org/10.1145/3009971

programmers' design efforts and to decrease the development time, recent research has focused on automating the transformation from high-level languages (such as C/C++) to RTL implementations [Villarreal et al. 2010; Cong et al. 2011; Canis et al. 2013; viv 2012] as well as providing high-level communication and memory abstractions [Chung et al. 2011; Adler et al. 2011; Yang et al. 2014; Fleming et al. 2014].

Among its many benefits, high-level abstraction separates algorithm design from the implementation details of the FPGA platform, such as memory, communications, and other ancillary service layers. FPGA programmers can therefore concentrate on algorithm design while programming against a fixed interface layer. Underneath this layer, system developers and compilers have enormous flexibility to generate platform service implementations optimized for the target application. In addition, as FPGAs have grown in size and capacity, FPGA programs do not usually consume all the resources available on a given FPGA, partly due to design difficulty such as cache scalability and partly due to design reuse, especially when the design is ported from an older, smaller FPGA to a newer, larger FPGA. There is significant opportunity for compilers to exploit the unused resources to optimize the underlying platform implementation. For example, a compiler might increase the amount of buffering in communication channels or increase the size of the on-chip storage to improve throughput.

In both processors and FPGAs, the memory system is usually critical to overall application performance. While the memory system on a specific conventional processor is fixed, cache algorithms and memory hierarchies on FPGAs can be tailored for different applications at compile time. For example, a well-pipelined, throughput-oriented program might not need a cache at all, while a latency-oriented program, like a soft-processor or a graph algorithm implementation, might prefer a low-latency first-level cache. Construction of program-optimized memory systems has three requirements. First, we need a memory abstraction that separates 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 a rich set of memory building blocks, such as caches with different latency, capacity, and associativity, from which a compiler can construct a tuned implementation. Finally, we need intelligent algorithms to analyze the program's memory access characteristics and automatically compose a hierarchy from the available building blocks.

Recent research has provided a number of abstract memory interfaces that enable compilers to assist in the construction of a program's memory system. For example, the Connected RAM (CoRAM) architecture [Chung et al. 2011] defines an application environment that separates computation from memory accesses managed by control threads, which fetch data from off-chip memory to on-chip buffers using a C-like language. LEAP (Latency-insensitive Environment for Application Programming) private memories [Adler et al. 2011] provide a memory abstraction with a simple request-response interface and manage a memory hierarchy from on-chip block RAMs (BRAMs) to the host memory. We adopt the LEAP private memory abstraction as the base of our work, because it provides a simple user interface, while giving us enough flexibility to construct a diverse set of memory hierarchies.

In this work, we present Scavenger, which targets one aspect of constructing program-optimized memories. Scavenger aims to improve program performance by utilizing spare memory resources left by the user program when building on-chip caches in the memory hierarchy. A key contribution of Scavenger is the exploration of microarchitectures for large on-chip caches on FPGAs. 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 build on-chip caches with

large capacity, we provide a scalable on-chip storage primitive by splitting a large BRAM structure into multiple BRAM banks, trading additional access latency for maintenance of high operating frequency. 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. We also evaluate a multi-word, set-associative structure for the shared cache, which can take advantage of spatial locality and reduce conflict misses among sharers at the cost of additional latency. To facilitate application-specific cache microarchitecture exploration, Scavenger provides a framework that enables programmers to easily configure multiple levels of caches through parameters. For example, programmers can easily enable cache banking, control the associativity of the shared cache, or disable some level of cache in the on-chip cache hierarchy if necessary.

Given this rich set of memory building blocks, the final step is to construct a hierarchy that is optimized for a particular program. Although Scavenger allows programmers to customize a memory hierarchy, we make some steps in the direction of automating this construction process. In specific, we introduce a two-phase memory system compilation flow into LEAP. The compiler first estimates the BRAM usage of the target user program and then automatically constructs a memory hierarchy tuned to use the remaining on-chip resources while maintaining the design frequency.

We evaluate the performance of our customized memory hierarchies in different workloads, including applications assembled by hand and applications compiled through high-level synthesis (HLS). We also study the cache design tradeoffs for each application. Compared to HLS-compiled applications, hand-assembled applications are well pipelined and thus less sensitive to memory latency, while HLS-compiled applications are more latency sensitive and benefit more from memory system improvements. We find that using banked BRAM structures to scale private caches provides 7% to 74% performance improvement (with a 26% geometric mean), over the baseline cache microarchitecture. Compared to the implementation with a minimal cache configuration, constructing maximal scalable caches achieves a 2.63× performance gain on average.

This manuscript is an extended version of the work published in Yang et al. [2015]. Our contributions in Yang et al. [2015] can be summarized as follows:

—We proposed a multi-cycle banked BRAM structure to construct scalable caches (Section 4.2).
—We designed an on-chip shared cache (Section 4.1), which can be automatically added to the memory hierarchy during compilation to consume most of the unused BRAM resources (Section 5.2).
—We provided a framework that allows programmers to configure a multi-level on-chip cache hierarchy through parameters, facilitating the exploration of cache microarchitectures.
—We evaluated the performance of our scalable cache microarchitectures and the on-chip shared cache across a set of benchmarks (Section 6).

In this article, we extend the previous work in the following ways:

—We propose a new cache indexing mechanism to support caches with non-power-of-two sizes, enabling our cache implementations to maximize resource utilization (Section 4.3).
—We apply our scalable BRAM structures to caches in LEAP coherent memories [Yang et al. 2014] and evaluate the performance of our scalable coherent caches with a shared memory application (Section 6).

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

Fig. 1. A general memory interface for hardware designs.

—We examine the impact of large on-chip caches on power and energy efficiency by measuring the power consumption on the FPGA and on the on-board dynamic random-access memory (DRAM) (Section 6).

2. BACKGROUND

Our exploration of memory system architecture builds on prior work in the development of FPGA memory systems: LEAP private memories [Adler et al. 2011] and coherent memories [Yang et al. 2014].

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

LEAP coherent memories [Yang et al. 2014] extend the baseline LEAP memory interface to support shared memory. Similarly to LEAP private memories, LEAP coherent memories provide a simple memory interface and the illusion of unlimited virtual storage. In addition, LEAP coherent memories permit applications to declare multiple, independent coherent address spaces using a compile-time specified domain name. LEAP coherent memories that share the same address space maintain cache coherence and provide a week consistency model. Fence operations are used to provide stronger consistency guarantees.

LEAP's memory system resembles that of general-purpose machines, both in terms of its abstract interface and its hierarchical construction. Like the load-store interface of general-purpose machines, LEAP's abstract memory interfaces do not specify or imply any details of the underlying memory system implementation, such as how many operations can be in flight or the topology of the memory. LEAP memory interfaces are also latency insensitive, which means it may take an arbitrary number of cycles to receive a memory response. 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. Any FPGA programs in which memory latency can only affect performance but not functionality can use LEAP memories. LEAP exploits abstraction to build complex, optimized memory architectures on behalf of the user, bridging the simple user interface and complex physical hardware.

At compile time, LEAP gathers various 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 shows an example of a typical LEAP memory hierarchy that integrates one private memory and three coherent memories instantiated in the user program. LEAP coherent memories are built on top of LEAP private memories both literally and figuratively: Each LEAP shared

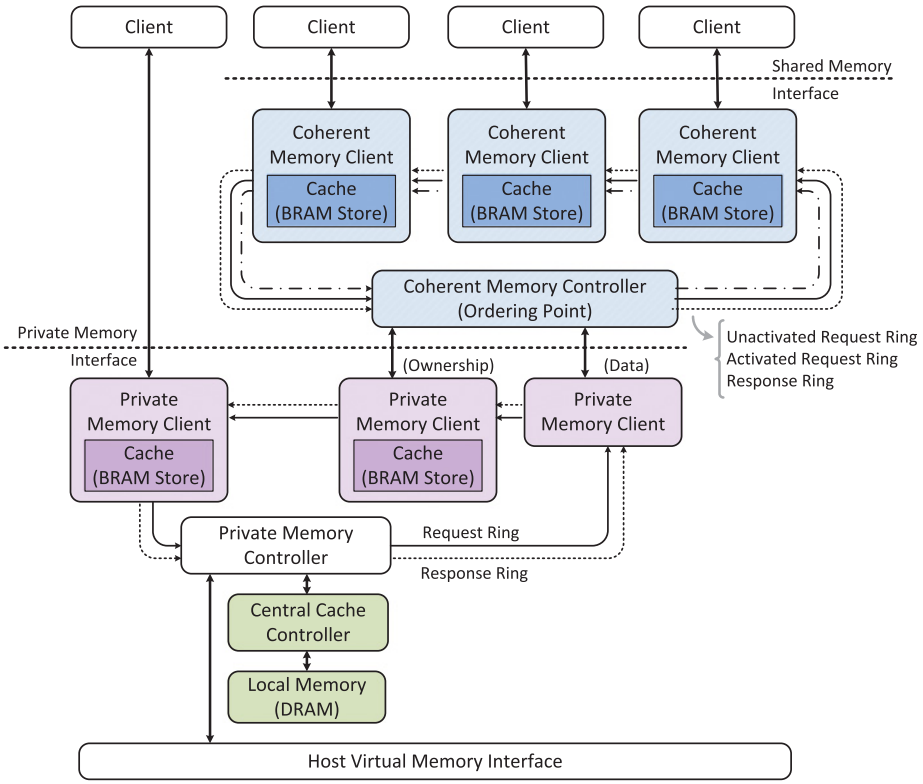


Fig. 2. An example of LEAP memory hierarchy.

address space is backed by two LEAP private memories as data and coherence ownership stores. As a result of this layering, both types of memories share large portions of the memory hierarchy.

LEAP memory clients optionally receive a local cache, which is direct mapped and implemented using on-chip SRAMs. For coherent memory clients in the same coherence domain, a snoopy-based coherence protocol is implemented to maintain cache coherency among their local caches. The board-level memory, which is typically an off-chip SRAM or DRAM, is used as a shared cache or central cache. The central cache controller manages access to a multi-word, set-associative board-level cache with a configurable replacement policy. Within the cache, each private memory space and shared memory domain is uniquely tagged, enforcing a physical separation. The program-facing caches are connected by way of a compiler-synthesized interconnect network. The main memory of an attached host processor backs this synthesized cache hierarchy.

Scavenger extends the scope of the LEAP memory system to enable better generation of program-optimized memory systems. The LEAP memory hierarchy provides a basic configuration interface that enables programmers to manipulate various parameters in the cache hierarchy. For example, the programmer may change the size of the cache or aspects of the cache allocation policy, within the scope of a two-level cache hierarchy. We add several new cache implementations, mostly targeting the construction of very large BRAM caches and introduce a second-level on-chip shared cache. We also extend the LEAP memory control interface, allowing programmers to configure a multi-level on-chip cache hierarchy and explore design tradeoffs for both private and shared

on-chip caches. Finally, we extend the LEAP compilation flow to include some limited automation for program-specific cache construction.

3. RELATED WORK

Much recent work has gone into the microarchitecture of shared caches on FPGAs [Dessouky et al. 2014; Choi et al. 2012] and the construction of FPGA-based multiple-level memory hierarchies [Adler et al. 2011; Choi et al. 2012; Göhringer et al. 2011; Matthews et al. 2015], all of which use off-chip storage for the second-level memory. Unfortunately, none of these works explore the design space of large or second-level on-chip caches, even though this is a common technique in processor architecture.

Large on-chip cache architectures have been extensively studied in the field of processor architectures [Kim et al. 2002; Agarwal et al. 2003]. Kim et al. [2002] 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. [2003] reduce cache access delay to meet target frequency by pipelining and aggressively banking the cache. In Scavenger, 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. We study cache design tradeoffs among access latency, operating frequency, and cache capacity for several FPGA applications with different memory access behavior. We are also not aware of any work that explores the microarchitectural tradeoffs of large on-chip caches on the FPGA.

Dessouky et al. [2014] propose DOMMU, 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 Scavenger, client regions are disjoint among the processing elements. A key difference between Scavenger 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 terms of both the number of PEs and BRAMs it can support and operating frequency. Scavenger 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. [2012] incorporates the multiporting approach of LaForest and Steffan [2010] into a cache hierarchy composed of a shared, multiported L1 cache backed by an off-chip RAM. Although the multiporting technique of LaForest 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. Scavenger's high-level architecture is similar to Choi, but Scavenger adopts a more scalable microarchitecture for its caches. Matthews et al. [2015] also explores L1 cache microarchitecture, focusing on set associativity and replacement policy in addition to multiporting. Like Scavenger, Matthews finds that increasing cache complexity, especially with respect to cache sizing, tends to decrease operating frequency, although the relative decline reported is smaller than the decline in large Scavenger caches.

We also explore indexing algorithms in Scavenger for caches with non-power-of-two sizes. Existing non-power-of-two indexing algorithms, such as the prime number indexing schemes [Lawrie and Vora 1982; Gao 1993] and Arbitrary Modulus Indexing [Diamond et al. 2014], 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 Scavenger, we construct caches with non-power-of-two sizes for the purpose 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 work presupposes the existence of a basic memory abstraction for FPGAs. We build on the LEAP memory framework, but there are other frameworks to which our work is equally applicable. CoRAM [Chung et al. 2011] advocates memory interaction using control threads programmed with a C-like language. CoRAM does not define the memory hierarchy backing its programmer interface and therefore could make use of our cache infrastructure. Similarly, FPGA-based processor infrastructures [Matthews et al. 2012; Lange et al. 2011; Mirian and Chow 2012] could also make use of our memory hierarchy.

4. SCALABLE MEMORY PRIMITIVES

In general-purpose processors, architects must 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 differs. Memory system designers can choose a memory system per application. 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.

The first goal of this work is to provide a richer set of memory primitives from which memory systems may be constructed. The second goal is to automate the construction of application-optimized memory systems from this richer set of primitives. This section describes the scalable memory primitives we provide. In Section 5, we will discuss how we use these primitives to construct memory systems.

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 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:

$$\text{performance} \propto \frac{f}{\max(n_c, n_{hit} + r \cdot n_{miss})}, \tag{1}$$

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, we propose two different approaches that utilize extra BRAM resources: (1) scaling the size of the first-level caches to decrease the miss

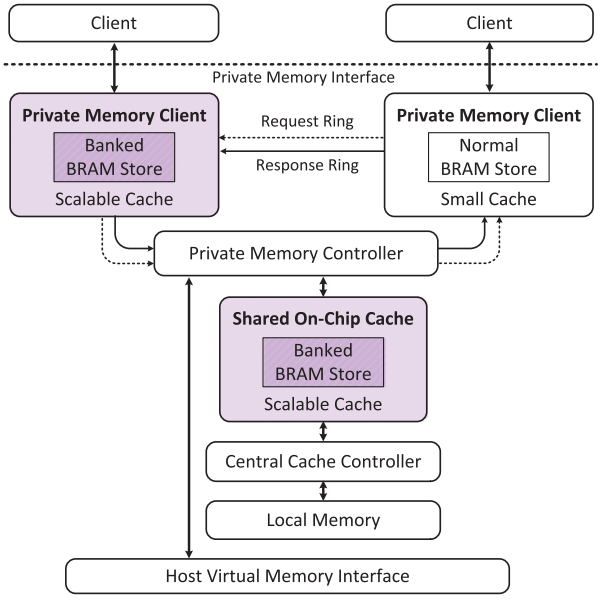


Fig. 3. LEAP private memory hierarchy extended by Scavenger, including scalable BRAM caches and a shared on-chip cache. Our modifications are highlighted.

rate (r) and (2) 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 naïvely 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 4.2. In addition, to completely exploit BRAM resources, Section 4.3 describes how we construct caches with non-power-of-two sizes.

4.1. On-chip Shared Cache

In LEAP's baseline memory hierarchy, as shown in Figure 2, BRAM resources are consumed by the first-level caches in LEAP private memories and coherent memories. If there are multiple memory clients, then 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 sub-program may have a dynamically variable working set size or different runtime memory footprint. 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 processing engines.

Figure 3 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. Similarly 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.

In order to reduce conflict misses, the on-chip shared cache is set associative. 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 the number of ways, can

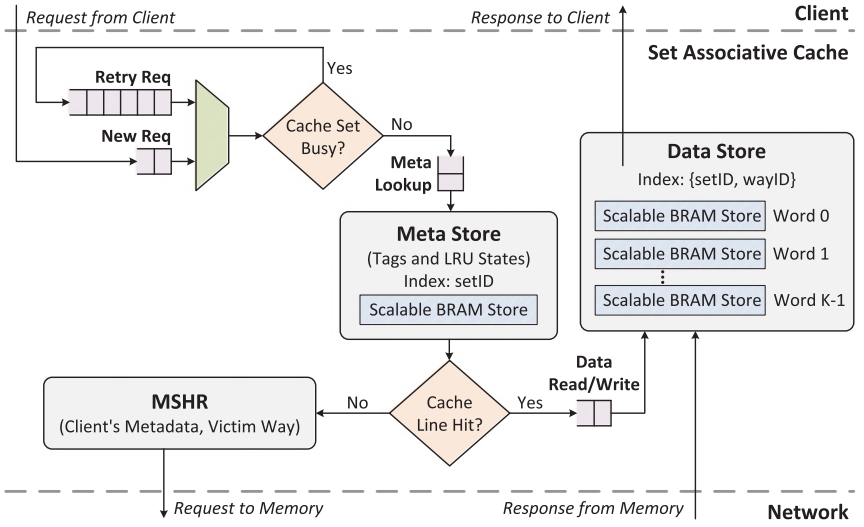


Fig. 4. Set associative cache microarchitecture.

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, which is direct-mapped, has one word per cacheline to reduce the area utilization of a replicated structure, the on-chip shared cache stores multiple words per cacheline to take advantage of spatial locality and to better utilize backing store bandwidth, since theses stores usually have larger line sizes.

Figure 4 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 cachelines, 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 cachelines from a particular way. To compensate the extra access latencies introduced by deeper 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 4, this cache structure can be built with our scalable BRAM store described in Section 4.2. 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.

4.2. 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 8KB memory, then there is little utility in scaling

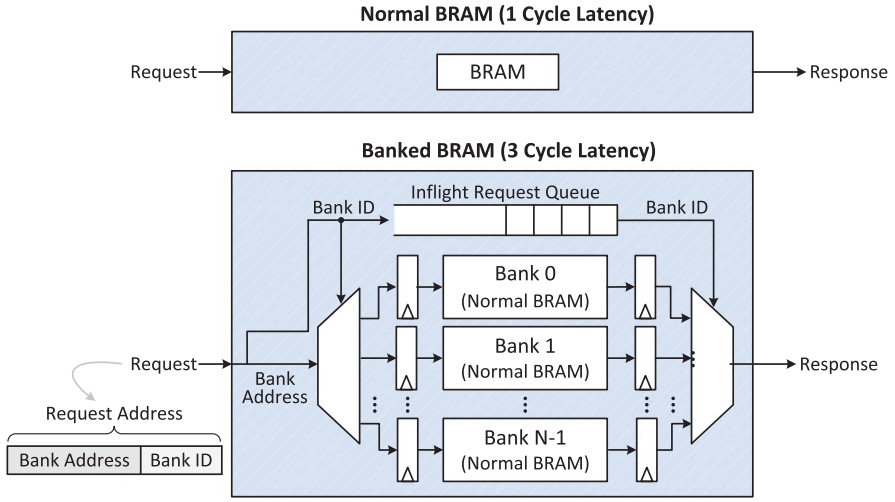


Fig. 5. BRAM store implementation option.

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 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 the responses from each bank can be reordered for return to the client.

Similarly to the on-chip shared cache described in Section 4.1, 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.

4.3. 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 6.1 and Table I, for many benchmarks we studied, there are around 35% BRAM resources left unused when building first-level caches with maximal sizes under power-of-two sizing limitations.

To completely utilize available BRAM resources, we need to eliminate power-of-two sizing restrictions. Constructing set-associative caches (as described in Section 4.1) 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,

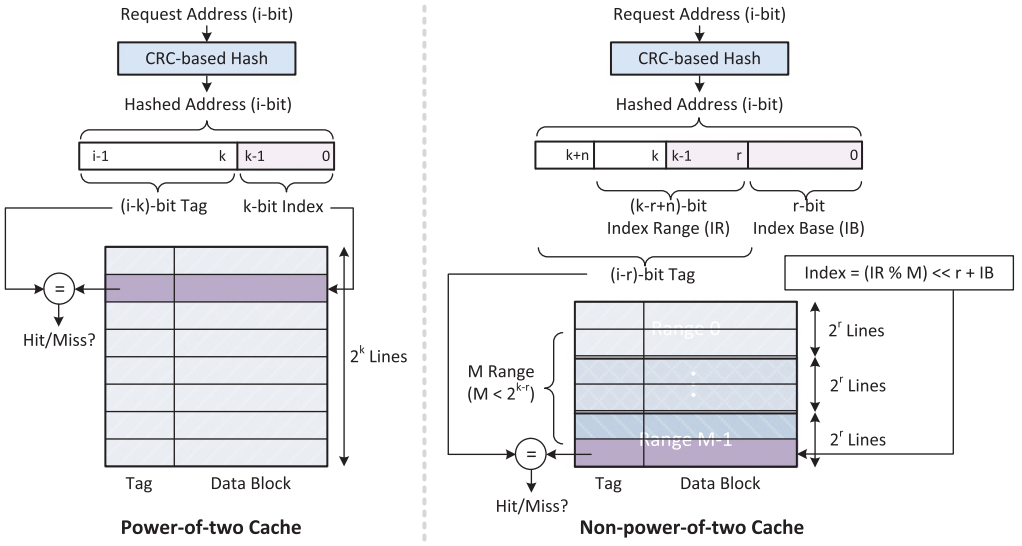


Fig. 6. Cache indexing mechanisms for power-of-two and non-power-of-two caches.

first-level caches, we instead modify the existing cache indexing mechanism for direct-mapped caches.

Figure 6 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 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 cachelines, 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 $M \cdot 2^r$ cachelines 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:

$$\text{Index} = (\text{IR} \bmod M) \ll r + \text{IB}. \tag{2}$$

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 $k-r$ 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. DESIGN TRADEOFF AND COMPILE TIME OPTIMIZATION

5.1. Design Tradeoff

As described in Equation (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.

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 $f_m > f_t$, then increase the cache capacity until it hits the frequency limitation.
(2) If $f_m \geq f_t$, then adopt the banked BRAM structure with longer cache latency and then increase the cache capacity until it hits the frequency limitation.
(3) If $f_m \leq f_t$, then decrease the frequency to $f_{t'}$ (where $f_{t'} \leq f_m$) in exchange for larger cache capacity.
(4) If $f_m \leq f_t$, then decrease the frequency to $f_{t''}$ (where $f_{t'} \leq f_{t''} \leq f_m$) 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 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. 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, the number of ways, and the type of BRAM storage. This database provides us with implementation feasibility information so 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 7 shows an example of a program using LEAP private memories and built with the two-phase compilation. Programs using LEAP coherent memories can

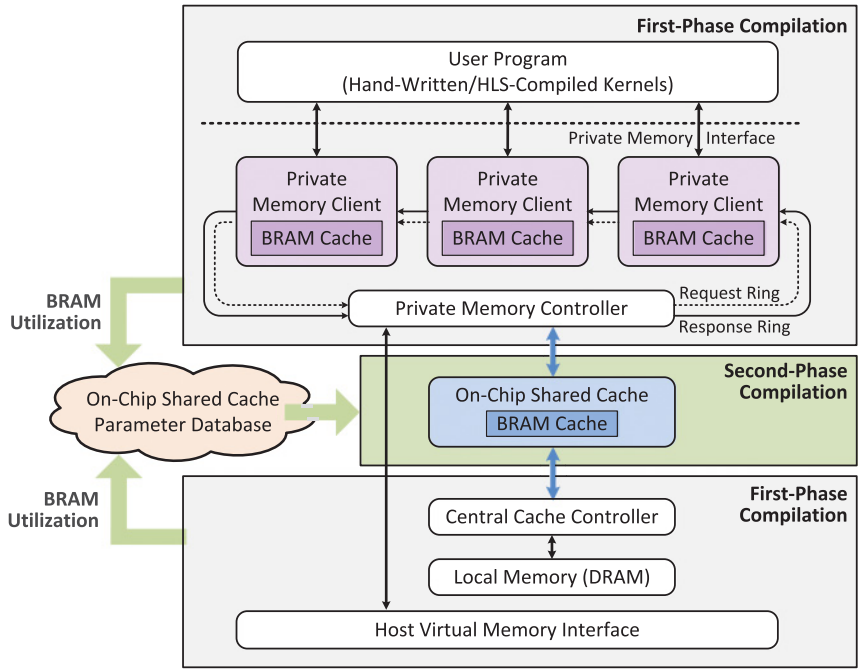


Fig. 7. An example of LEAP-based program built with the two-phase compilation flow.

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, then the compiler will use the pre-built database to greedily select the largest cache that can be implemented while using the remaining BRAM at the target frequency. If there are multiple candidates with the same cache capacity, then 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, then 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 image while reusing most of the synthesis results obtained in the first phase.

6. EVALUATION

We evaluate all of our test programs on the Xilinx VC707 platform. Our VC707 deployment includes a 1GB DDR3 memory, which we use to implement the off-chip cache. For HLS benchmarks [Winterstein et al. 2014], we utilize Vivado HLS [Vivado 2012]. 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.

We examine a set of benchmarks ranging from kernels to large programs that nearly fill the VC707:

Memperf: A kernel measuring performance of the LEAP memory hierarchy by testing various data strides and working set sizes on a single private memory.

Heat: A two-dimensional stencil code that models heat transfer across a surface. *Heat* is embarrassingly parallel and can be divided among as many work engines as can be fit on the FPGA. *Heat* is also very regular: Workers march over the shared two-dimensional space in fixed rectangular patterns. In the single-worker implementation, the worker accesses a single private memory. In multi-worker implementations, each *heat* worker accesses a single coherent memory.

HAsim: *HAsim* [Pellauer et al. 2011] 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: caches, branch prediction tables, and virtual-address translation buffers. The largest private memory used by *HAsim* captures the virtual memory state of the modeled processors. The *HAsim* results in this section are obtained by scaling the first-level cache in this largest private memory.

Merger: An HLS kernel that merges several linked lists together to form a sorted list. *Merger* 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.

Prio: An HLS kernel implementing a priority queue using a sorted, doubly linked list. The application performs a sorted insertion for each new entry. Unlike *merger*, *prio* maintains a single queue and uses one private memory, allowing us to observe the effect of a single cache on memory access latency.

Filter: An HLS kernel that implements a filtering algorithm for k-means clustering [Kanungo et al. 2002]. k-means clustering partitions a data set of points into k clusters, such that each point belongs to the cluster with the nearest mean. *Filter* first builds a binary tree structure from the input data set and then traverses the tree in several iterations. Our implementation splits the tree into eight independently processed sub-trees. Each partition tracks its tree traversal using a stack and maintains several sets of candidates for the best cluster centers. *Filter* uses 24 LEAP private memories: eight each for the sub-trees, stacks, and candidate center sets. The *filter* results in this section are obtained by scaling all first-level caches in the system. Although *filter* uses many LEAP memories, only the working set of tree nodes is large, and only these memories benefit from cache capacity scaling.

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 I lists the BRAM utilization for the various benchmarks examined in this study. Generally, the BRAM utilization for the user program is quite low: Most of the applications that we study are computational kernels. Even *HAsim*, which is a heavy consumer of logic resources for larger core-count

Table I. Post-Place-and-Route BRAM Utilization and Frequency Results

Program	User-Kernel BRAM Usage (%)	Scavenger Total BRAM Usage (%)			Post P&R f_{max} (MHz)				
						max (2^k)		max	
		min	*max* (2^k)	*max*	*min*	*banked*	*monolithic*	*banked*	*monolithic*
Memperf	0	4	61	90	150	140	80	140	70
HAsim (16 cores)	6	11	73	91	110	110	75	110	75
Heat (1 worker)	0	4	61	95	150	140	80	140	70
Heat (8 workers)	0	7	69	69	125	110	100	110	100
Merger	17	19	75	75	140	140	90	140	90
Prio	19	25	80	96	140	135	90	130	80
Filter	7	18	90	90	140	140	110	140	110

Note: BRAM utilization results are reported as a fraction of the total BRAM resources for the VC707. For each application, we report the user-kernel BRAM usage to give an idea of the amount of BRAMs available for platform optimizations in Scavenger. To show the impact large caches have on operating frequency, we report the total area utilization and maximum achievable frequency when building with minimal and maximal first-level cache size configurations, where *max* (2^k) denotes the maximal cache size configuration limited by the power-of-two sizing rule.

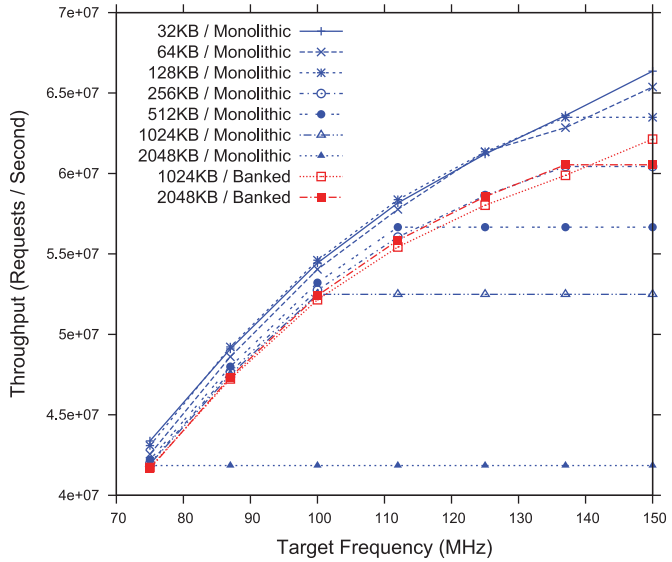


Fig. 8. L1 cache performance with various microarchitectures. Flat lines indicate that target frequency is not met.

models, uses relatively little BRAM. 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 BRAM in support of the memory hierarchy. Under the power-of-two cache sizing limitation, there are still some BRAMs left after the caches are scaled to the maximal sizes, especially for single-memory applications, such as *memperf*, single-worker *heat*, and *prio*. Without the cache sizing limitation, Scavenger can utilize more than 90% BRAMs for many applications.

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 8 shows a study of cache size and microarchitecture using *memperf* with the stride equal to 1 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 75MHz to 150MHz, 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 8, a monolithic 2MB cache, which uses more than 60% of the BRAM available on the VC707, runs at 75MHz, while a four-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. 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 8 shows that we can maintain frequency for a small kernel. In general, our large banked caches do not incur a significant frequency penalty. Table I 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.

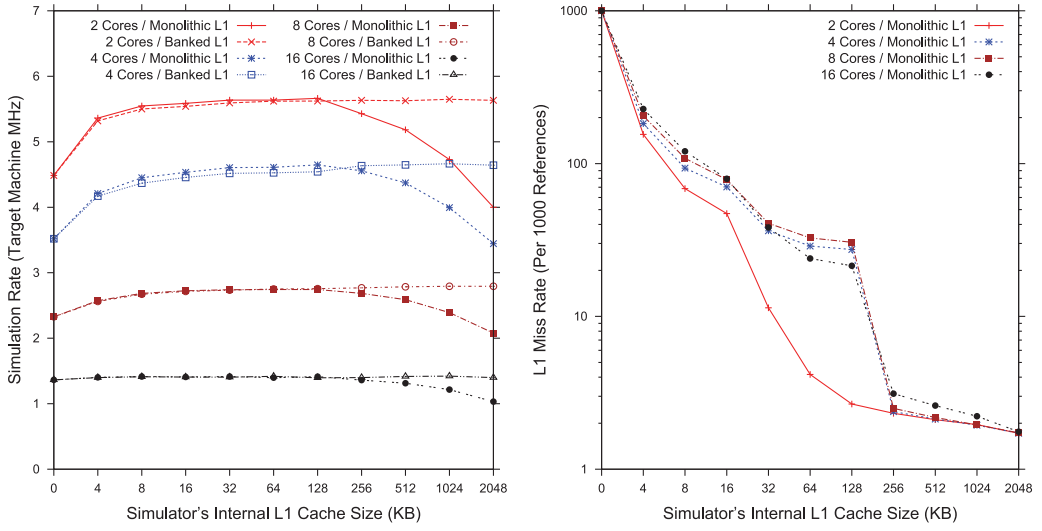
6.3. Application Performance with Large L1 Caches

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 running at the maximal achievable frequency. We will show that maximal memory configurations do not always deliver the best performance.

HAsim: Intuitively, a processor model should obtain performance gains from larger caches, just as processors do. Figure 9(b) 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 maximum of 4%, as shown in Figure 9(a). This limited gain is a result of the deep pipelining of the *HAsim* model. Once a small first-level cache filters enough requests to reduce the bandwidth of requests to backing caches, *HAsim* 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.

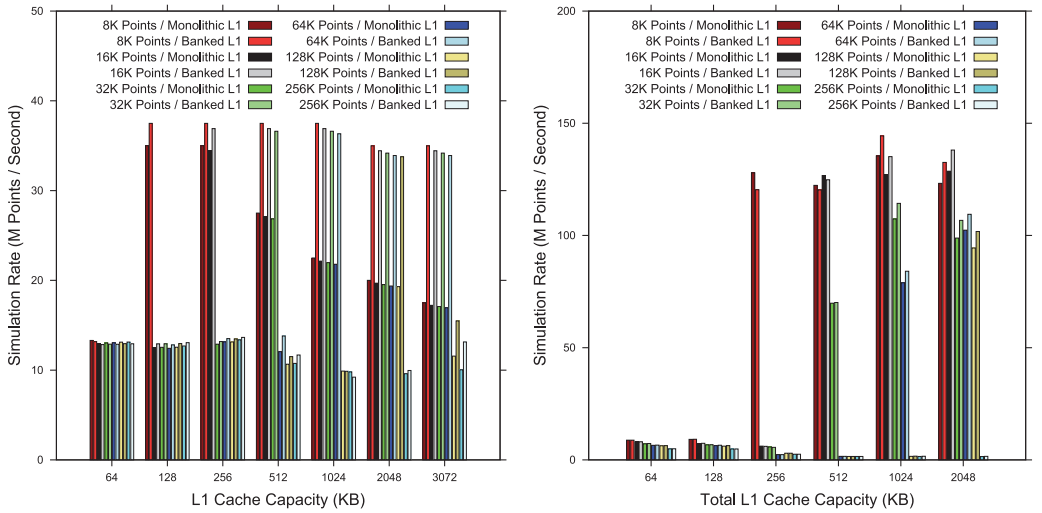
Because *HAsim* is latency tolerant, trading any frequency for cache capacity is a loss on the VC707 (thus options 1 and 2 in Section 5.1 are preferred). *HAsim's* maximum operating frequency tops out at 110MHz, enabling us to use our largest banked cache. Thus, although our banked caches do not help *HAsim* much, they do not harm it either. On the other hand, large, monolithic caches degrade *HAsim's* performance by up to 30%.

Heat: Figure 10 shows the performance of single-worker and multi-worker *heat* with various cache configurations and problem sizes. Generally, the performance of the *heat* stencil is determined by the block size that can be fit into cache. If the problem size is



(a) Performance (b) Cache Miss Rate

Fig. 9. Performance metrics for *HAsim* with various cache configurations. Performance drops as simulated core count grows because all cores are simulated in a shared, multiplexed pipeline. Although large caches dramatically improve miss rate, this does not translate to a large performance gain.



(a) Single-worker Implementation (b) 8-worker Implementation

Fig. 10. Performance comparison for *Heat* with various cache configurations.

too large to fit into cache, then *heat* will always miss to the nearest memory sufficient to hold its working set. As we scale our caches, the larger *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 the single-worker version, the frequency penalty of monolithic microarchitecture

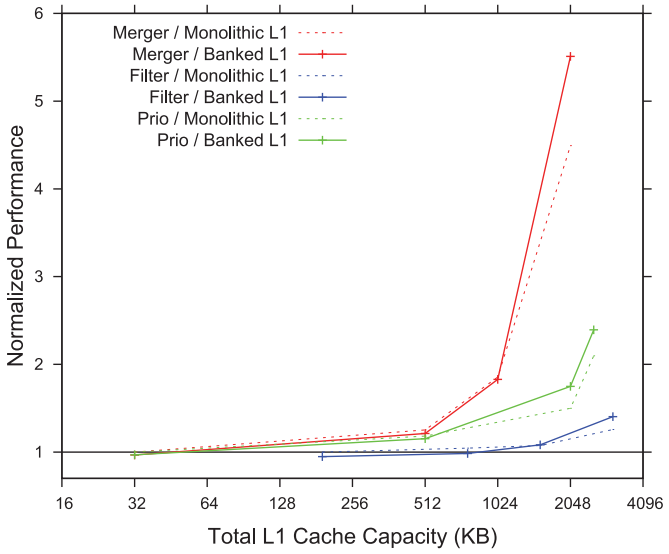


Fig. 11. Performance comparison for HLS kernels with various L1 cache sizes. Performance results are normalized to the performance of the smallest monolithic cache.

is lower, causing a smaller performance difference between banked and monolithic caches. With our banked microarchitecture, the maximum operating frequency of single-worker *heat* declines less than 10%, even when implementing a *two megabyte* cache. Due to complicated data paths in coherent cache implementations, multi-worker *heat* runs at slightly lower frequencies and the frequency penalties of large caches are higher. *Heat* is fairly sensitive to latency in the memory system: 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 5.1); for large problem sizes, having large banked cache and slightly downgrading the frequency (option 4 in Section 5.1) achieves the best performance gain.

HLS Kernels: In Figure 11, we explore scaling the capacity of the L1 caches for our HLS kernels. These applications use BRAM 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-die resources. Using our approach, *prio* is able to use 96% of the on-chip BRAM resources.

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× 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× to 2.39× with BRAM utilization increased from 80% to 96%. 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 5.1).

Summary: Figure 12 summarizes the performance improvement achieved by building largest L1 caches with the banked BRAM structure. When constructing L1 caches with

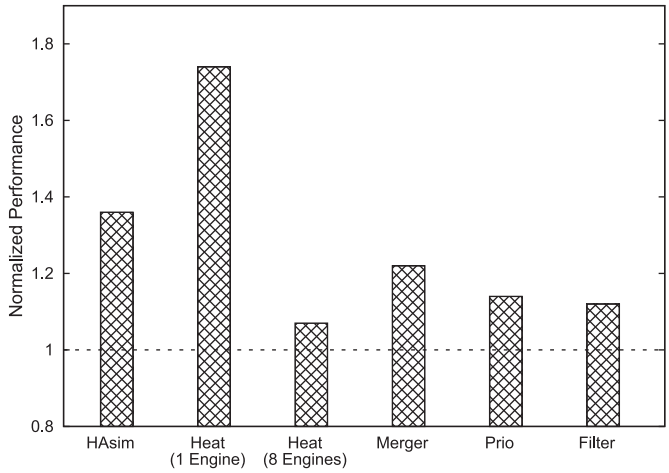


Fig. 12. Performance of the banked cache for various benchmarks built with maximal L1 cache sizes. For each benchmark, the banked cache performance is normalized to the monolithic cache with the same cache size. A geometric mean is used for *HAsim* and *Heat*, which have multiple problem size configurations.

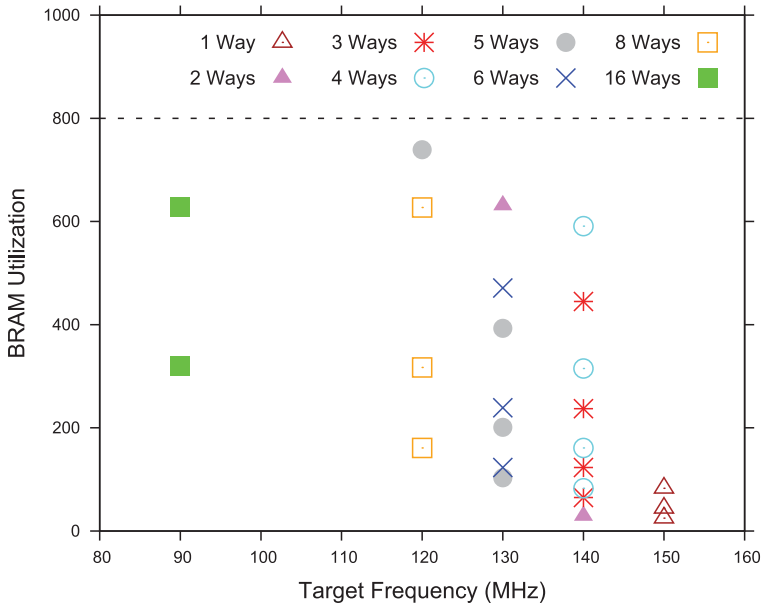
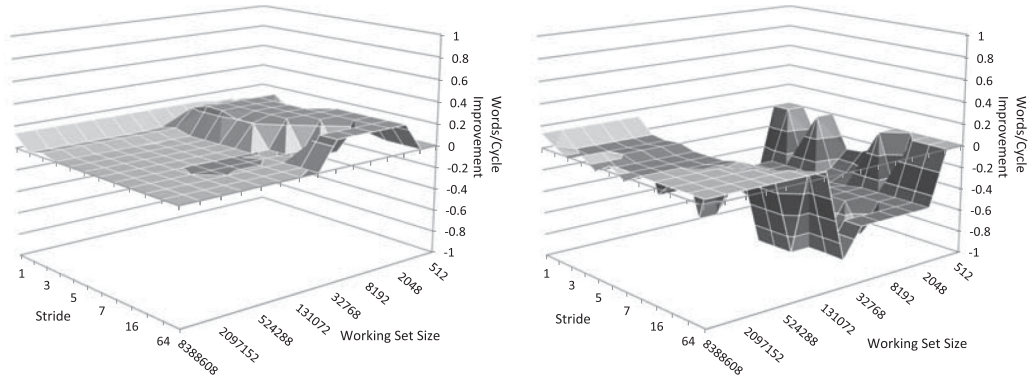


Fig. 13. Implementation space of L2 caches.

maximal sizes, our banked cache, which runs at higher frequencies, provides 7% to 74% performance gains (with a 26% geometric mean) over the baseline monolithic cache.

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 5.1). Figure 13 describes



(a) Automatically-sized L2 Cache vs. Baseline LEAP memory implementation.

(b) L2 Cache vs. Equal Capacity L1

Fig. 14. Relative throughput gain for *memperf*.

Table II. L2 Cache Performance Gain for *Merger*

	BRAM Usage (%)		Runtime (s)		
Total L1 Cache Size (KB)	L1 Only	L1 + L2	L1 Only	L1 + L2	Performance Gain (%)
32	18.7	52.3	803.8	601.1	33.7
512	32.5	64.5	642.1	534.4	20.2
1024	46.8	80.8	433.2	362.4	19.5
2048	75.1	93.3	145.9	136.9	6.6

how we choose the size of this L2 cache. By exploiting set associativity, we are able to cover a broad 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 14(a) shows the result of running our L2 cache selection algorithm on *memperf*. For this benchmark, our algorithm selects a four-way cache with 8,192 sets, utilizing about 33% of the BRAM available on the chip. For those working sets that fit in the L2 cache, performance improves by about 25%. For completeness, in Figure 14(b), we compare our algorithm against a direct-mapped L1 with equal area (128K words). The large L1 handily outperforms our L2 for those sizes that 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 that do not fit in the cache. This is because the L2 captures some spatial locality within the stride sets.

Table II 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 four-way L2 cache with 4,096 sets, improving performance by 6.6%. The performance gain comes from higher associativity and multi-word cachelines that capture a degree of spatial locality in the L1 miss stream.

6.5. Energy Consumption

The goal of Scavenger is to utilize spare BRAM resources for constructing application-specific cache hierarchies that are optimized for minimum program execution latency.

Table III. Power and Energy Measurements for HLS Applications

Total L1 Cache Size (KB)	Performance Speedup	P_{DRAM} (W)	P_{FPGA} (W)	E_{TOTAL} (J)
Merger (Input: 4*79990 random non-negative integers)				
32	1.00	1.75	1.89	2,926
512	1.25	1.75	2.36	2,643
1,024	1.86	1.55	2.67	1,827
2,048	5.51	1.03	3.84	711
Prio (Input: 327676 random non-negative integers)				
32	1.00	1.18	1.74	37,074
512	1.18	1.16	2.25	36,660
2,560	2.39	1.01	3.53	24,134
Filter (32759 tree nodes, 128 clusters, 256 iterations)				
192	1.00	1.00	2.38	54
768	1.05	1.01	2.84	59
1,536	1.08	1.02	3.83	71
3,072	1.40	1.01	5.17	70

Note: For a given cache size, we choose the cache microarchitecture that achieves better runtime performance and report the performance speedup against the performance of the system with a minimal cache size. For each test, we also record the average power consumption of the FPGA (P_{FPGA}) and the DRAM (P_{DRAM}) as well as the total energy consumption (E_{TOTAL}).

In Section 6.3 and Section 6.4, we have shown that Scavenger is able to achieve up to 5.9× speedup by constructing scalable caches. However, these performance-optimized solutions may not always achieve the best energy efficiency.

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. Table III shows the average FPGA and DRAM power consumption as well as the total energy consumption 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 III, 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.

7. CONCLUSION

High-level abstractions provide FPGA compilers flexibility to customize platform implementations for different applications without perturbing the user program. Scavenger demonstrates that it is possible to exploit unutilized resources to construct memory system implementations that accelerate the user program. The space of potential memory hierarchies is large and the problem of deciding how to build application-optimized memory hierarchies is difficult. By providing a large set of memory building blocks and a framework for integrating them, Scavenger enables experimentation in both of these spaces. In this work, we have focused primarily on the exploration of the memory hierarchy space, while making a small step toward automating the construction memory hierarchies based on program resource utilization and frequency requirements.

Since many programs leave large amounts of memory resources unused, our memory building blocks tend to be quite large. We propose microarchitecture changes for large on-chip caches to run at high frequency. We had good success in building large first-level caches. Across our benchmark suite, we obtained performance gains of 7% to 74% (with a 26% geometric mean) over the baseline cache microarchitecture.

In addition, we provide a set-associative shared cache structure that can be automatically constructed by the compiler. Although 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 first-level private caches, at least for the applications that 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 memory system on FPGA can be customized with large first-level 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, throughput-oriented 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.

The distribution of unused on-chip resources across a memory hierarchy to optimize program performance likely requires sophisticated program analysis. Our simple, greedy shared-cache allocator yielded limited gains in part due to its simplicity and in part due to the ineffectiveness of the shared cache on our applications. In this work, we make no claim of finding optimal solutions for the cache topology and sizing. A main contribution is to provide beneficial directions in the design space exploration and to provide a framework that facilitates such microarchitectural exploration. Future work will use both static and dynamic program analysis to optimize the resource distribution in the memory hierarchy. We also expect to be able to make decisions about cache associativity, cache replacement policy, and the inclusion of advanced microarchitectural features like prefetching. In addition, we intend to further experiment with cache microarchitecture, for example, placing caches in separate clock domains to alleviate timing pressure. We also plan to investigate our caching scheme in the context of power consumption and consider energy-performance tradeoffs when deciding program-optimized cache hierarchies.

REFERENCES

Michael Adler, Kermin Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. LEAP scratchpads: Automatic memory and cache management for reconfigurable logic. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

Amit Agarwal, Kaushik Roy, and T. N. Vijaykumar. 2003. Exploring high bandwidth pipelined cache architecture for scaled technology. In *Design, Automation Test in Europe Conference Exhibition (DATE)*.

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.* 13, 2 (2013), 24.

J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. 2012. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An in-fabric memory abstraction for FPGA-based computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 30, 4 (2011), 473–491.

G. Dessouky, M. J. Klaiber, D. G. Bailey, and S. Simon. 2014. Adaptive dynamic on-chip memory management for FPGA-based reconfigurable architectures. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

Jeffrey R. Diamond, Donald S. Fussell, and Stephen W. Keckler. 2014. Arbitrary modulus indexing. In *International Symposium on Microarchitecture (MICRO)*.

Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. 2014. The LEAP FPGA operating system. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

Q. S. Gao. 1993. The chinese remainder theorem and the prime memory system. In *ACM SIGARCH Computer Architecture News*.

D. Göhringer, L. Meder, M. Hübner, and J. Becker. 2011. Adaptive multi-client network-on-chip memory. In *International Conference on Reconfigurable Computing and FPGAs (RECONFIG)*.

Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 7 (2002), 881–892.

Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Charles Eric LaForest and J. Gregory Steffan. 2010. Efficient multi-ported memories for FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

H. Lange, T. Wink, and A. Koch. 2011. MARC ii: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In *Design, Automation Test in Europe Conference Exhibition (DATE)*.

Duncan H. Lawrie and Chandra R. Vora. 1982. The prime memory system for array access. *IEEE Trans. Comput.* 31, 5 (1982), 435–442.

Eric Matthews, Nicholas C. Doyle, and Lesley Shannon. 2015. Design space exploration of L1 data caches for FPGA-based multiprocessor systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

E. Matthews, L. Shannon, and A. Fedorova. 2012. Polyblaze: From one to many bringing the microblaze into the multicore era with linux SMP support. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

Vincent Mirian and Paul Chow. 2012. FCache: A system for cache coherent processing on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. 2011. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. 2010. Designing modular hardware accelerators in c with ROCCC 2.0. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Vivado. 2012. Vivado High-Level Synthesis. Retrieved from http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2014. Separation logic-assisted code transformations for efficient high-level synthesis. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Hsin-Jung Yang, Kermin Fleming, Michael Adler, and Joel Emer. 2014. LEAP shared memories: Automating the construction of FPGA coherent memories. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Hsin-Jung Yang, Kermin Fleming, Michael Adler, Felix Winterstein, and Joel Emer. 2015. Scavenger: Automating the construction of application-optimized memory hierarchies. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

Received April 2016; revised August 2016; accepted October 2016