Scavenger: Automating the Construction of Application-Optimized Memory Hierarchies

Hsin-Jung Yang*, Kermin Fleming†, Michael Adler†, Felix Winterstein‡, Joel Emer*§
*CSAIL, Massachusetts Institute of Technology
†SSG, Intel Corporation
‡Ground Station Systems Division, European Space Agency
§Architecture Research Group, NVIDIA Corporation
Email: *{hjyang, emer}@csail.mit.edu, †{kermin.fleming, michael.adler}@intel.com, ‡f.winterstein12@imperial.ac.uk

***Abstract*—**

High-level abstractions separate algorithm design from platform implementation, allowing programmers to focus on algorithms while building increasingly complex systems. This separation also provides system programmers and compilers an opportunity to optimize platform services for each application. In FPGAs, this 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 often 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 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 BRAM structures, we provide microarchitectural techniques to trade memory latency for design frequency. We demonstrate, by examining both hand-assembled and HLS-compiled benchmarks, that our application-optimized memory system can improve pre-existing application runtime by 25% on average.

I. Introduction

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 programmers' design effort 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 [1] [2] [3] [4] as well as providing high-level communication and memory abstractions [5] [6].

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 platform layer. Underneath this layer, system developers and compilers have enormous flexibility to generate platform service implementations optimized for the target application. Since FPGA programs do not usually consume all the resources on a given FPGA, there is significant opportunity to exploit the unused resources to optimize the platform implementation. For example, a compiler might allocate extra functional units to increase algorithm parallelism or it might increase the scale of buffering in the design to improve throughput.

For many applications, both in processors and in FPGAs, the memory system is critical to overall performance. While the memory system on a specific conventional processor is fixed, cache algorithms or the memory hierarchies on FPGAs can be tailored for different applications. 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 fast first-level cache. Our goal is to build the best possible memory system for each target application. 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, CoRAM [5] proposes a cache hierarchy in which programmers write control threads that fetch data from memory to on-chip buffers using a C-like language. LEAP Scratchpads [6] provide a memory abstraction with a simple request-response interface and manage a memory hierarchy from on-chip BRAMs to the host memory. We adopt LEAP scratchpad's memory abstraction for our work, because it provides a simple user interface, while giving us enough flexibility to construct a diverse set of memory hierarchies.

A key contribution of this work is the exploration of microarchitectures for large on-chip caches. FPGA-based caches are built by aggregating distributed on-die BRAM resources. 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 split large BRAM structures into multiple BRAM banks, trading additional latency for maintenance of high operating frequency. In addition, we extend LEAP's memory hierarchy to include an on-chip shared cache which, for typical designs, can consume most unused BRAM. 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 to take advantage of spatial locality and to reduce conflict misses among sharers.

```
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.

Given this rich set of memory building blocks, the final step is to construct a hierarchy that is optimal for a particular program. We make some steps in the direction of automating this construction process by introducing a two-stage 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 the customized memory hierarchies in different applications, both hand-assembled and HLS-compiled. 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 25% runtime improvement, on average, over the baseline implementation.

II. Overview

Our exploration of memory system architecture builds upon prior work in the development of FPGA memory systems: LEAP Scratchpads [6]. LEAP Scratchpads provide a general, in-fabric memory abstraction for FPGA programs. Programmers instantiate memories with the 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 have arbitrary type and arbitrary size, even if the target FPGA does not have sufficient physical memory to cover the entire requested memory space.

Like the load-store interface of general-purpose machines, LEAP's abstract memory interface does not specify or imply any details of the underlying memory system implementation, such as how many operations can be in flight and the topology of the memory. This ambiguity provides significant freedom of implementation to the compiler. For example, a small memory could be implemented as a local SRAM, while a larger memory could be backed by a cache hierarchy and host virtual memory. LEAP leverages this freedom 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 the various scratchpads in the user program and instantiates a complex memory hierarchy with multiple levels of cache, as in Figure 2. Scratchpad memories instantiated in the user program optionally receive a local cache, which is a direct mapped cache with a standard eviction policy. The board-level memory, typically an off-chip SRAM or DRAM, is used as a shared cache. 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. Like memory hierarchies in general-purpose computers, scratchpads provide the appearance of fast memory to programs with good locality,

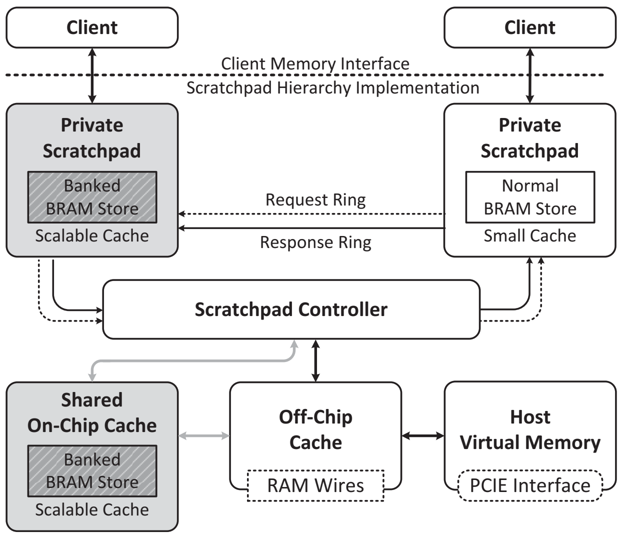


Fig. 2: LEAP Scratchpad Memory Architecture. Client interfaces are backed by a compiled memory hierarchy. In this work, we extend that memory hierarchy to support very large BRAM caches. We further extend the memory hierarchy to include a shared, chip-level BRAM cache. Our modifications are shaded to differentiate them from the baseline hierarchy.

while maintaining the abstraction of a large address space through the virtual memory mechanisms of the host.

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 control interface which enables the programmer 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 cache. Finally, we extend the LEAP compilation flow to include some limited automation for program-specific cache construction.

III. Related Work

Much recent work has gone in to the microarchitecture of shared caches on FPGAs [7] [8] and the construction of multiple level-memory hierarchies [6] [8] [9] [10], all of which use off-chip storage for the L2 memory. Unfortunately none of these works explore the design space of large or L2 caches, even though this is a common technique in processor architecture. We are not aware of any work that explores the microarchitectural tradeoffs of large on-chip caches in FPGA.

Dessouky et. al. [7] 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

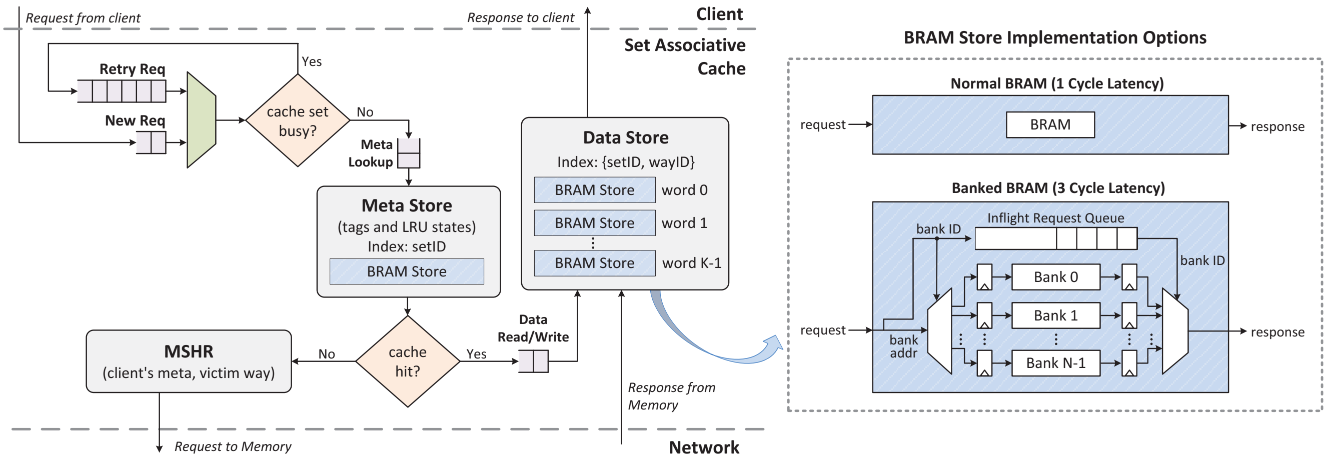


Fig. 3: Set associative cache microarchitecture with scalable BRAM stores. The set associative cache microarchitecture is shown on the left, with choices of store on the right.

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 [8] incorporates the multiporting approach of LaForest [11] into a cache hierarchy comprised of a shared, multi-ported 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. [10] 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.

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 [5] 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 [12] [13] [14], could also make use of our memory hierarchy.

IV. Scalable Memory Primitives

In general purpose processors, architects must fix the memory system parameters at design time. Parameters like cache size, 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. 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 V, we will discuss how we use these primitives to construct memory systems.

For many applications, program performance is affected by the operating frequency and the latency of the underlying memory hierarchy. To reduce the memory latency, we propose two different approaches that utilize extra BRAM resources: (1) scaling the size of the private caches to increase the hit rate, and (2) adding a large mid-level shared BRAM cache sitting on top of the off-chip cache to reduce the miss penalty of the private caches. Section IV-A describes our on-chip shared cache implementation. 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 IV-B.

A. On-chip Shared Cache

In LEAP's baseline memory hierarchy, as shown in Figure 2, BRAM resources are consumed by the caches in private scratchpads. If there are multiple memory requesters, partitioning and distributing on-chip memory optimally can be challenging. Consider a program containing different types of processing engines and each connecting to a private scratchpad. Constructing symmetric private 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. Since scratchpads are caching private data, there is no need for coherence in this configuration.

In order to reduce conflict misses, the on-chip shared cache is designed to support associativity. In the case that multiple scratchpads 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 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 private 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. Figure 3 shows the microarchitecture of the set associative cache. Following the common design choices of building last level caches in processors, we store metadata and actual data values separately and read them serially for power and timing optimizations. 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. This cache structure can be built with our scalable BRAM store described in Section IV-B. 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.

B. 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. Generally, this difficulty arises because FPGA programs are explicit in their use of memory: if a program asks for a physical 8KB memory, there is little utility in scaling this memory. However, even abstract FPGA programs have difficulty in utilizing new memory 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 BRAM latency for maintaining frequency and high capacity. Figure 3 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 address 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. Since the control logic and the BRAM store in the cache are separable, we are free to combine them and form a new cache implementation, as shown in Figure 3.

V. Design Tradeoff and Compile Time Optimization

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 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 $f_m > f_t$, 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 $f_{t'} \leq f_{t''} \leq f_m$) and increase the cache latency (using banked BRAM) in exchange for larger cache capacity.

Option 1 preserves performance but has limited scalability. This can be adopted for programs that do not need 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 VI, 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.

To automatically utilize the BRAM resources that are not consumed by user designs, 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. This database provides us with implementation feasibility information so that our program modifications minimize impact on user operating frequency. Then we adopt a two-phase compilation flow. During the first phase of compilation, the scratchpad memory hierarchy is constructed based on the user's parameter choices or default settings if no choices have been made. After the first phase, the compiler gathers the BRAM usage information from RTL synthesis. The compiler then uses the pre-built database to greedily select the largest cache that can be implemented while using the remaining BRAM at the target frequency. If the amount of remaining BRAM resources is sufficient to meaningfully increase the size of the cache, the compiler will configure and build an optimized memory system during the second phase of compilation, while reusing most of the original synthesis results.

VI. Evaluation

We evaluate all of our test programs on the Xilinx VC707 platform. Our VC707 deployment includes a 1GB DDR2 memory, which we use to implement the off-chip cache. For HLS benchmarks [15], we utilize Vivado HLS. We make use

Program	User 18K-bit BRAM	Scavenger 18K-bit BRAM		Post P&R f_{max} (MHz)		
		min	*max*	*min*	*max*	
					banked	*monolithic*
Memperf	0%	8%	58%	150	135	75
HAsim (8 cores)	14%	11%	58%	110	110	75
Heat	2%	2%	58%	150	140	80
Merger	14%	4%	64%	135	120	80
Prio	16%	2%	63%	135	110	60
Filter	5%	6%	87%	120	115	75

TABLE I: Post-place and route design characteristics for various programs. BRAM utilization results are reported as a fraction of the total BRAM resources for the VC707. For each application, we report the area utilization and frequency for a minimal and maximal private cache size configuration, to give an idea of the impact large caches have on operating frequency.

of Xilinx Vivado 2014.4 for all physical implementation work, and most synthesis work, with the exception of caches with non-power-of-two ways and our HLS kernels. Due to a minor bug in the Xilinx tools, we use Synplify 2013.09 for these caches. All resource 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 scratchpad.

Heat: A two-dimensional stencil code which 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. Each *heat* worker uses a single scratchpad.

HAsim: *HAsim* [16] 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 private scratchpads to model various structures of the processor: caches, branch prediction tables, and translation buffers. The largest memory used by *HAsim* captures the virtual memory state of the modeled processor.

Merger: An HLS kernel that merges several linked lists together to form a sorted list. *Merger* forms four linked lists in parallel form streams of random integers. After a constant number of input values 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 private scratchpads.

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 scratchpad, 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 [17]. K-means clustering partitions a data set of points into K clusters, such that each point belongs to the cluster with the nearest mean. *Filter* first builds a binary tree structure from the input data set and then traverses the tree in several iterations. Our implementation splits the tree into eight independently-processed sub-trees. Each partition tracks its tree traversal using a stack and maintains several sets of candidates for the best cluster centers. *Filter* uses 24 private scratchpads: eight each for the sub-trees, stacks, and candidate centers sets. Although *filter* uses many scratchpads, only the working set of tree nodes is large, and only these scratchpads benefit form capacity scaling.

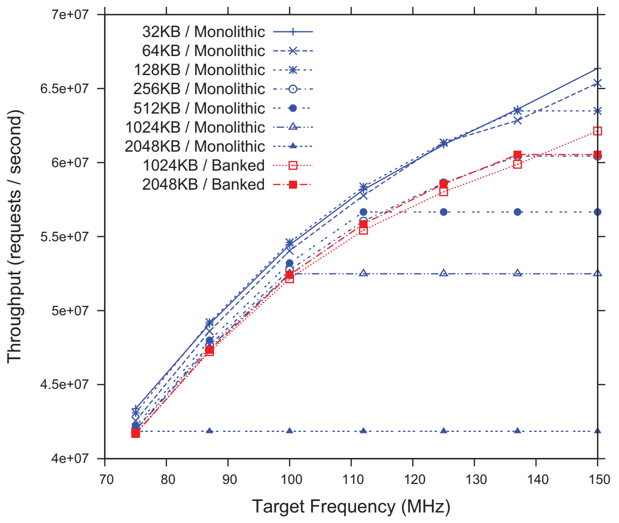


Fig. 4: L1 cache performance with various microarchitectures. Flat lines indicate that target frequency was not met.

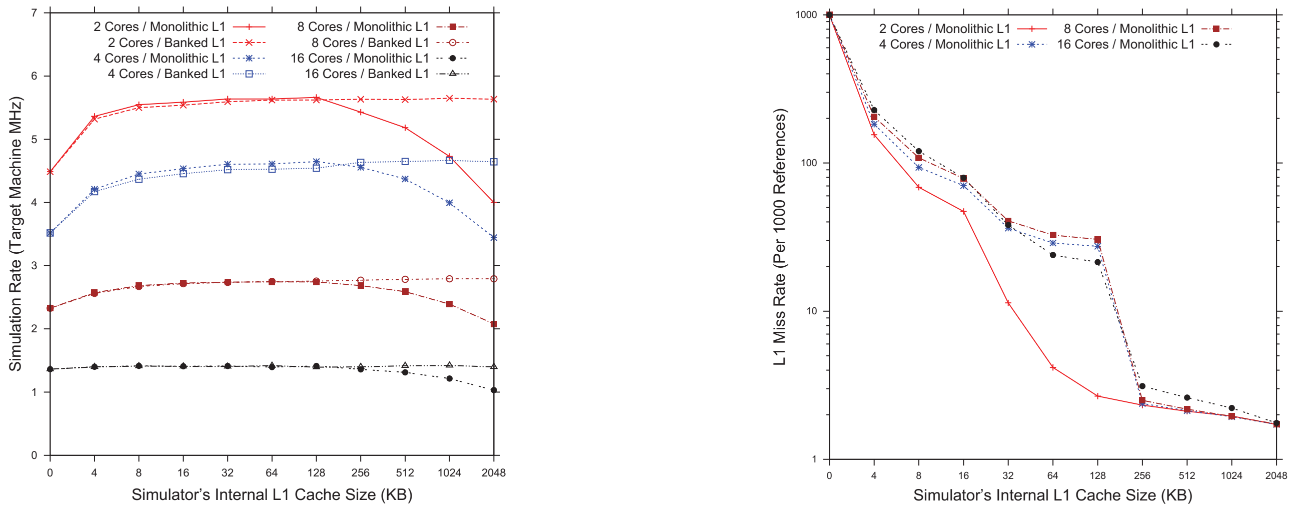
A. BRAM Utilization

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 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.

B. 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 4 shows a study of cache size and microarchitecture using *memperf* with stride equal to one and with working set equivalent to cache capacity. In this figure, 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. A monolithic 2MB cache, which uses more than 60% of the BRAM available on the VC707, runs at no more than 75 MHz, while a 4-way banked cache achieves nearly twice that frequency. The timing relaxation afforded by



(a) Performance (b) Cache Miss Rate

Fig. 5: 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.

our buffered architecture even enables individual banks to clock faster than similarly sized monolithic caches.

In theory, all of our caches should have the same performance for *memperf*. In practice, cache performance is variable within a small range. Because we use a hashing scheme to improve overall cache performance, the number of L1 conflict misses at 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 4 shows that we can maintain frequency for a small kernel. In general, our large caches do not incur a significant frequency penalty. Table I lists the operating frequency for each application with a smallest and a largest cache size configuration. The table includes the frequencies of the maximal memory system configurations implemented with monolithic and banked BRAM stores. The upper bound of BRAM usage for maximal memory configurations are mostly around 60% due to the power-of-two scaling in our first-level caches. With banked stores, even when using most of the on-die BRAM resources, we suffer a frequency penalty no larger than 20%. Note that maximal memory configurations do not always deliver the best performance. Compared to monolithic BRAM stores, banked BRAM stores achieve up to 80% frequency gain when building large caches.

C. Application Performance with Large L1 Caches

HAsim: Intuitively, a processor model should obtain performance gains from larger caches, just as processors do. Figure 5b bears this intuition out: *HAsim's* cache misses drop dramatically as we scale cache size and capture larger portions of the model working set. However, this improvement in cache miss rate does not translate into absolute performance. Rather, *HAsim's* performance with respect to cache size rises until the cache reaches a medium size and then plateaus, rising by a maximum of 4%, as shown in Figure 5a. 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 V are preferred). *HAsim'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 *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 6 shows the performance of *heat* with various cache configurations. Generally, the performance of the *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 *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. For very large caches, our banked microarchitecture outperforms the baseline monolithic cache by 79%, due to high operating frequency. With our banked microarchitecture, the maximum operating frequency of *heat* declines less than 10%, even when implementing a *two megabyte* cache. *Heat* is fairly sensitive to latency in the memory system: at low frequencies, monolithic caches slightly outperform banked caches. For smaller problem sizes, having smaller, but fast caches is preferred (option 1 in Section V); for large problem sizes, having large banked cache and slightly downgrading the frequency (option 4 in Section V) achieves the best performance gain.

HLS Kernels: In Figure 7, we explore scaling the capacity of the L1 caches for our HLS kernels. These applications use

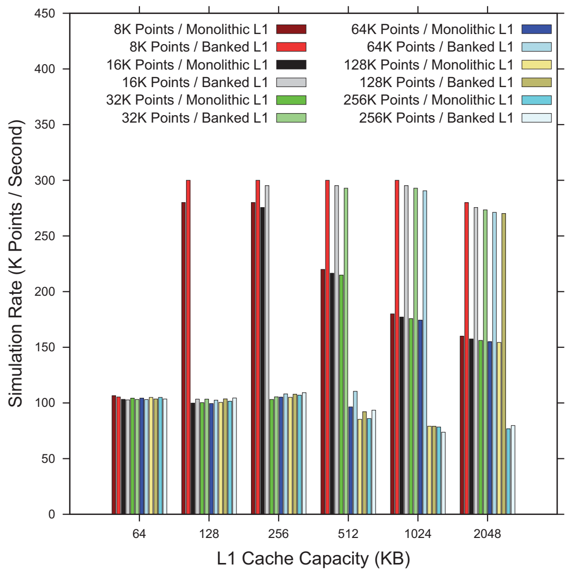


Fig. 6: Resource utilization and performance comparison for heat transfer. Heat experiences excellent frequency scaling.

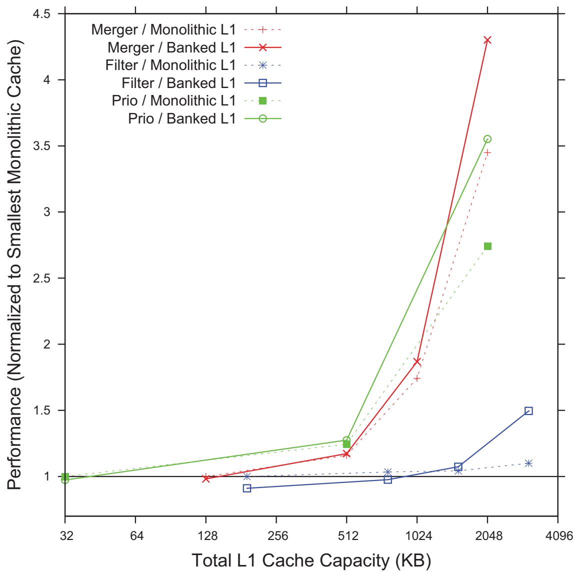


Fig. 7: Performance comparison for HLS kernels.

L1 Cache Size	Runtime (ms)		Performance Gain (%)
	L1 Only	L1 + L2	
8kB	2703	2422	10.4
32kB	2614	2501	4.3
64KB	2518	2342	7.0
128KB	1807	1744	3.5

TABLE II: L2 cache performance gain for filtering algorithm.

BRAM internally, but use less than 20% of the resources on the chip. Our cache structures enable these applications to gain some benefit from the remaining on-die resources. Using our approach, *filter* is, remarkably, able to use 96% of the on-chip BRAM.

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 4x 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 cache size scales, the superior operating frequency of the banked cache results in a 20% absolute performance gain over the monolithic cache. 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 V).

D. *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 V).

Figure 8 describes 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 9 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 available on the chip. For those working sets that fit in the L2 cache, performance improves by about 25%.

Table II shows the result of applying our algorithm to *filter*. When the L1 cache size is fixed, adding an L2 cache always improves performance, although *filter* prefers large private scratchpads to a large shared cache. Yet, even when building largest feasible L1 caches, there remain 7% unused BRAM resources, which can be used to implement a small L2 cache (a direct-mapped cache with 4096 lines), improving performance by 3.5%. For *filter*, performance is improved because the small L2 captures a degree of spatial locality in the L1 miss stream by storing multiple words per cache line.

VII. 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, the cache building blocks of our memory system 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.

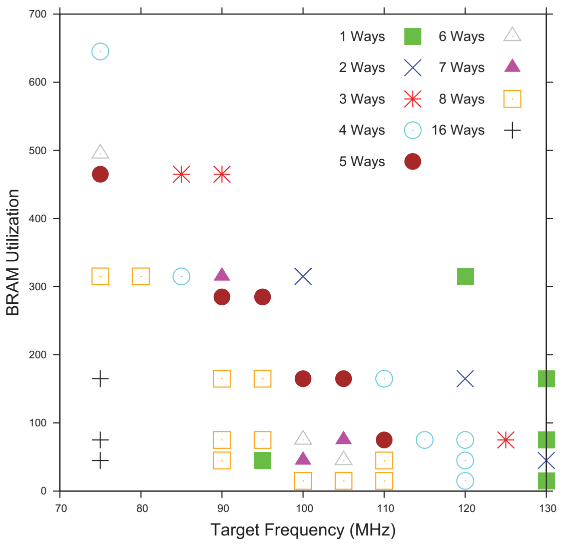


Fig. 8: Implementation space of L2 caches.

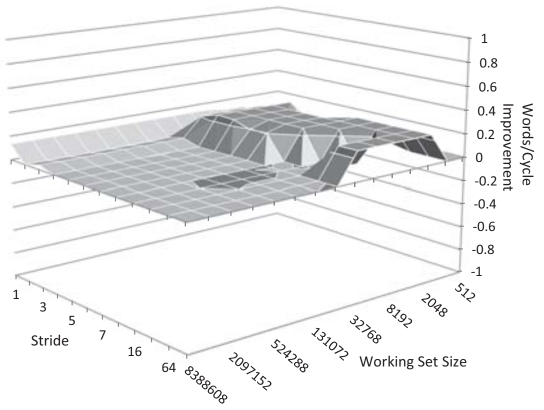


Fig. 9: Relative throughput gain from the automatically-sized L2 cache.

Across our benchmark suite, we obtained an average maximum performance gain of 25%

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 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. 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.

REFERENCES

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

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[4] "Vivado High-Level Synthesis," *http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html*.

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

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

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

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

[9] D. Goehringer, L. Meder, M. Hubner, and J. Becker, "Adaptive multi-client network-on-chip memory," in *ReConFig*, 2011.

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

[11] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *FPGA*, 2010.

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

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

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

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

[16] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *HPCA*, 2011.

[17] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, 2002.