OPTIMIZING UNDER ABSTRACTION: USING PREFETCHING TO IMPROVE FPGA PERFORMANCE

Hsin-Jung Yang[¶] *Kermin Fleming*[¶] *Michael Adler*[†] *Joel Emer*[†¶]

[¶]Computer Science and A.I. Laboratory
Massachusetts Institute of Technology
{hjyang, kfleming, emer}@csail.mit.edu

[†]VSSAD Group
Intel Corporation
{michael.adler, joel.emer}@intel.com

ABSTRACT

In an effort to speed the development of FPGA-based accelerators, recent research has focused on providing FPGA developers with memory and communications abstractions. Because abstraction decouples the function of these interfaces from their implementation, these new interfaces present an enormous opportunity for optimization. In this paper we examine stride prefetching as a means of improving the performance of an automatically synthesized, abstract memory hierarchy. We demonstrate, by applying our technique to several large benchmarks, that prefetching can improve preexisting application runtime by 15% on average, and up to 40%, without requiring program modification.

1. INTRODUCTION

FPGAs were originally intended to provide a replacement for ASICs in small or low-volume designs. However, as FPGAs have grown in both size and capability, they have matured from their original role to become algorithmic computation platforms in their own right. Indeed, many recent academic and industrial research projects have accelerated algorithms using FPGAs, without the intention of producing or emulating an ASIC. Instead, these projects targeted FPGAs to take advantage of the performance benefits offered by a fine-grained parallel substrate over general purpose processors. Rather than trying to precisely emulate some circuit, the design goal for these programs is to produce a correct answer to a problem of interest as quickly as possible.

This new use case for FPGAs has engendered several researches into making FPGAs easier to use, since generating an answer quickly encompasses not only the time to compute a solution but also the time to implement the program to perform the computation. The latter has traditionally been a pain point for FPGAs. FPGA tools, programming languages, and abstractions are lacking, prolonging development time and impeding the general adoption of FPGAs. In light of the need to make FPGAs truly programmable, much recent work has focused on making FPGAs easier to use.

One class of work that targets algorithm acceleration has added new programming primitives to existing hardware description languages(HDL). These augmentations simplify FPGA programming by permitting the HDL programmer to concisely express program behavior at a higher level than register transfers. Examples of higher-level constructs include communications, to software [1], within the FPGA, [2], and between FPGAs [3], and memory abstractions [4, 5]. The programmer contract of these primitives is *functional*, and in using these primitives, the programmer agrees to adhere to the function of the interface, rather than the timing of some specific instance of the interface. These primitives map well to FPGA-based algorithm accelerators: unlike RTL emulation, which requires the preservation of cycle accuracy, an algorithm accelerator requires only the preservation of the functional behavior of the algorithm.

In many cases, an algorithm implementation does not consume all the resources available on a given FPGA, for example, when an algorithm targeting an older, smaller FPGA is ported to a newer, larger FPGA. In this case, we would like the compiler to automatically make use of those resources to improve the performance of the program. To a limited extent, existing RTLs permit this kind of optimization. For example, re-timing and register duplication permit some scaling of program performance in exchange for area. However, in traditional RTLs, optimizations must operate at the sub-cycle level so as not to inadmissibly alter the cycle-level behavior of the RTL program. This constraint fundamentally limits the impact of traditional RTL optimizations on program performance. In contrast, the newer, abstract HDL primitives permit a great deal of freedom in terms of optimization: at points where the algorithm incorporates the new primitives, the compiler is explicitly free to choose any implementation that preserves the *functional* behavior of the HDL primitive. Thus, the compiler may leverage idle resources to construct a high-performance implementation underneath of the functional abstraction layer.

In this work, we examine automatically generated prefetching as a means of improving the performance of previously written FPGA applications. As in general purpose

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

processors, prefetchers can be introduced alongside the existing memory hierarchy without modifying the memory-using application. We present a novel, FPGA-optimized microarchitecture for prefetching which is tuned for the behavior of typical FPGA applications. We demonstrate the effectiveness of our approach by adding prefetchers to three large, preexisting FPGA programs. By making use of unused FPGA resources to automatically accelerate these programs, we obtain performance gains of 15% on average, and up to 40%, on unmodified program source.

2. RELATED WORK

Traditionally, the definition of FPGA memory hierarchy has been left as a sometimes-painful exercise to the programmer. In response to the increasing complexity of memory systems, FPGA vendors have begun to provide memory controllers, for example Xilinx MIG [6]. Programmers instantiate some physical memory controller, and tie their program logic directly to the instantiated controller, simplifying the overall design experience. Although controllers are upgraded by the vendors over time and maintain nearly constant interfaces across memory and FPGA generations, vendor memory controllers provide a basic implementation – advanced features like caches are still left to programmers.

Recent research into FPGA programming and architecture has suggested that programs can benefit, in some cases, dramatically, from improved memory system support. The CoRAM [5] architecture provides a C-like language in which fetches to memory can be described. These fetches are executed in a streaming fashion and stored in an SRAM resource for access by program logic. Control flow between the fabric and fetch thread permits a degree of program-dependent dynamic behavior in the fetch stream.

CoRAM is similar to both a cache hierarchy and a prefetcher. However, the chief weakness of CoRAM lies in the difficulty of describing the memory access pattern. CoRAM works well for deterministic access patterns, like matrix-matrix multiplication. However, for data-dependent or complex access patterns, a C-like fetch description language may be inadequate. For example, consider the strongly data-dependent access pattern of an H.264 decoder: memory accesses generation is tightly coupled to states and logic in the FPGA fabric, rendering it difficult to describe the fetch pattern in an external language.

An alternative approach is the LEAP Scratchpad memory architecture [4]. LEAP Scratchpads provide a general, portable memory abstraction for FPGA programs. Programmers instantiate memories with the simple read-request, read-response, write interface, shown in Figure 1. A program may instantiate as many Scratchpad memories as necessary, and these memories may have arbitrary size, even if the target FPGA does not have sufficient physical memory to cover the entire requested memory space.

At compile time, the compiler gathers the Scratchpads in the user program and instantiates a complex memory hierarchy [4] with multiple levels of cache, as in Figure 2. Scratchpad memories instantiated in the user program each receives a local L1 cache. The board-level memory, typically an off-chip SRAM or DRAM, is used as a shared L2 cache. The L1 caches are connected to the L2 by way of a synthesized interconnect network. Backing this high-performance cache hierarchy is the main memory of a host general-purpose computer. Like memory hierarchies in general-purpose computers, Scratchpads provide the appearance of fast memory to programs with good locality, while maintaining the illusion of an infinite store through the virtual memory mechanisms of the host.

```
interface MEM_IFC#(type addr, type data);
    Action readRequest(addr addrIn);
    ActionValue#(data) readResponse();
    Action write(addr addrIn, data dataIn);
endinterface
```

Fig. 1. A general memory interface for hardware designs [7].

The LEAP Scratchpad architecture provides FPGA programs a memory abstraction, enabling us to augment the memory hierarchy with optimization techniques similar to those used in general-purpose processors. Prefetching is one such technique. In processors, hardware prefetching mitigates the impact of cache misses by predicting program behavior and issuing memory reads in advance of program execution. Prefetching schemes attempt to predict two properties of data: *which* data will be used and *when* it will be used. Consequently, prefetching schemes are judged based on accuracy, fetching useful data, and timeliness, fetching data into the cache prior, but temporally close, to its use.

The simplest prefetching technique is next sequential prefetch, which issues a prefetch request to the next cache line ($L+1$) when the current cache line (L) is accessed [8, 9]. Prefetch-on-miss issues a prefetch request on every cache miss, while tagged-prefetch issues requests both on a cache miss and on the first access to a prefetched cache line, a prefetch hit, to further reduce the number of misses in a sequential access stream [10]. To enable tagged-prefetching, a tag bit is added to each cache line, marking prefetched cache lines that have not been accessed.

To enhance accuracy, hardware prefetching methods dynamically attempt to learn access patterns from program access streams. However, the learning process is complicated by the interleaving of streaming memory accesses with other streaming accesses and non-predictable data fetch. To separate different streaming accesses and to filter out the non-predictable accesses, extra information, such as the program counter (PC) of load instructions [11, 12] or the memory region of the target address [13, 14], is used to disambiguate the program memory access stream. Once memory access streams are separated, a prefetch learning algorithm [11, 14, 15, 16] can be applied to each stream independently. Stride-prefetching is the most common of these algorithms. When the prefetcher learns the stride pattern, it issues memory accesses for $L+s$, $L+2\cdot s$,..., $L+d\cdot s$, where L is the current cache line, s is the detected stride, and d is the prefetch degree, the number of issued prefetch requests. The d parameter is also called look-ahead distance [17], and may be adjusted to improve timeliness.

Recent prefetching schemes, such as Markov prefetching [15] and delta correlation prefetching [16], focus on detecting more complex memory patterns. To reduce the area complexity of these prefetchers, Global History Buffer (GHB) [18] uses shared memory structures and linked-lists to store long access histories. Diaz et al. [19] extend GHB by linking different local streams together to further increase accuracy and timeliness.

3. PREFETCHING IN FPGAS

Prefetching in general-purpose architecture has been well studied and widely applied in modern architectures. However, there are significant differences between prefetching in general-purpose architectures and prefetching on the FPGA, both in terms of program behavior and hardware implementation. By examining these qualitative differences we build intuition into our FPGA prefetching architecture design.

Hardware programs differ qualitatively from software programs. Hardware programs lack a PC, an important prefetching hint for general purpose processors. Although software prefetchers have access to the PC, the memory accesses streams produced by software programs are a mix of both data and program control structures, which are usually non-predictable even in streaming applications. On the other hand, the memory access streams produced by FPGA programs are, in many cases, pure data streams. In FPGA programs, control structures, which must be stored in memory in Von Neumann architectures, are stored in the fabric and accesses to these structures do not pollute the memory stream. Prefetchers in general-purpose processors rely on the PC to filter out non-predictable accesses, especially at the L1 cache. We show that reasonable prefetching is possible at the L1 cache in hardware programs even though the PC is not available.

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

The final difference between prefetching in processors and in FPGAs is in physical implementation of the prefetching hardware. Silicon prefetcher implementations have great freedom in building complex wired structures, including content-addressable memories (CAMs). However, general-purpose prefetchers face power-performance driven area constraints, resulting in small numbers of learners, particularly at lower levels of the cache. On the other hand, FPGAs have fixed resources on die, which must either be used by a program or left idle. Hardware programs running may leave a large portion of these resources unused, especially on large FPGAs. Therefore, the area constraints for prefetching algorithms on FPGAs are usually much lower than in a fixed-function processor design. Due to the nature of wires in FPGA fabric, CAMs are not well suited for FPGA implementation. Instead, larger direct-mapped structures, made efficient by plentiful in-fabric memory resources, are a good architecture for prefetching in the FPGA.

4. FPGA PREFETCHING MICROARCHITECTURE

Although FPGAs and general-purpose architectures are different, the similarity between the memory hierarchies in LEAP Scratchpads and general-purpose processors allows us to borrow the concepts of prefetch techniques used in processors to improve the performance of existing FPGA programs. In this section we describe our adaptation of classical prefetching techniques to the computational structures of FPGAs and the integration of our prefetching microarchitecture into the LEAP scratchpad memory hierarchy.

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

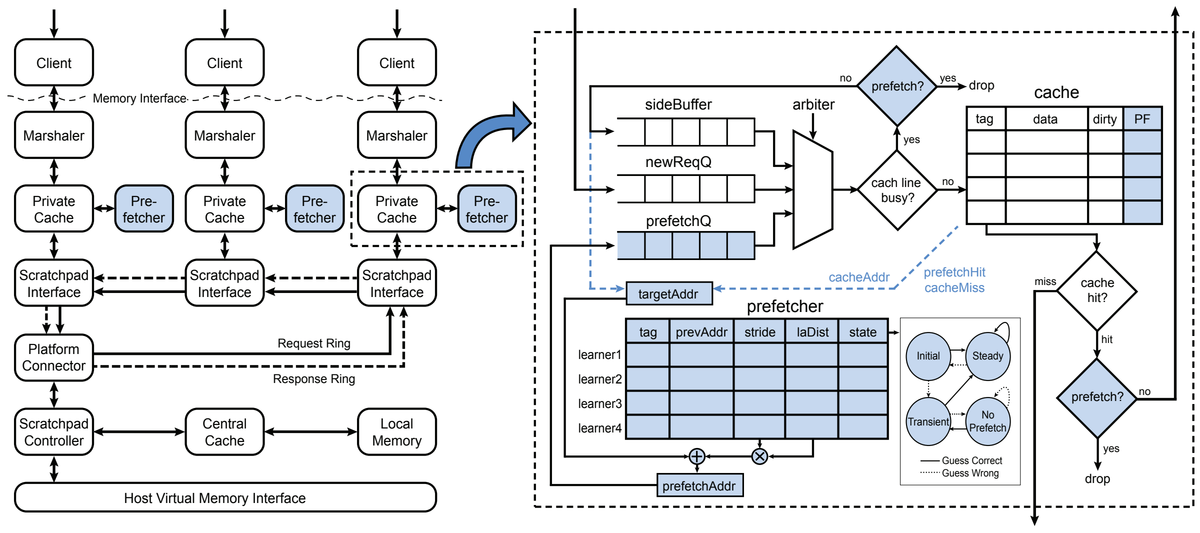


Fig. 2. Prefetching Microarchitecture. The LEAP Scratchpad architecture is shown to the left, with the L1 microarchitecture. Our augmentations are highlighted.

However, we believe that the address-based stride prefetching is sufficient to disambiguate multiple access streams because the memory accesses from hardware programs are pure data streams.

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

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

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

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

The left side of Figure 2 shows a modified Scratchpad architecture augmented with prefetchers. Each private L1 cache is connected to a prefetcher that learns from the cache accesses and issues prefetch requests to bring data into the cache. A prefetcher consists of multiple learners, each of which is responsible for extracting the stride pattern in a local memory access stream. The number of learners is parameterized and can be adjusted either by the programmer or by a compiler depending on the amount of available resources. Because we store the state of the learners in either FPGA LUTRAM or SRAM resources, we can instantiate many more learners on an FPGA board than in a processor, although these learners can only be accessed by direct address indexing. Indeed, because FPGA resources have a fixed minimum size, we fill the memory resource completely with learners – SRAM-based prefetchers may have thousands of learners. Using large numbers of learners approximates a fully associative structure and can significantly reduce conflict misses.

The right half of Figure 2 depicts the integration of our prefetching microarchitecture into the L1 cache microarchitecture. In the baseline scratchpad L1 cache microarchitecture, cache lines are either available for operation or are busy, if there is an outstanding request to an upper level cache. The baseline cache maintains two kinds of requests which are buffered in two distinct queues: (1) *newReqQ* buffers incoming client requests and (2) *sideBuffer* stores prior requests that are blocked by busy cache lines. Each cycle, an arbiter chooses a single request from one of the queues. If the chosen request needs to access a busy cache line, it is shunted to the *sideBuffer* where it waits for the busy line to be serviced. The *sideBuffer* allows subsequent cache requests to be processed out-of-order but requires the introduction of a completion buffer (not depicted) in the cache. Because out-of-order request processing obfuscates memory access patterns, our prefetcher learns only from the *newReqQ*.

In addition to the prefetching hardware itself, we make a few modifications to the scratchpad cache microarchitecture. Our prefetching microarchitecture adds a third request source to the request arbiter: *prefetchQ* buffers requests issued by the prefetcher. If a prefetch request is chosen and the request tries to access a busy cache line, it is not shunted to the *sideBuffer*, but instead dropped. Prefetches to busy lines are dropped because it is likely that the prefetch request is issued too late, and the cache line is busy because the client request predicted by the prefetcher has already occurred. In addition, we add an additional prefetch status bit (*PF*) to each cache line to mark cache lines that are pulled into the cache by prefetch requests and have not been accessed yet. This additional bit enables the tagged prefetching scheme as well as allows the prefetching hardware to check whether a prefetch request is timely or not.

5. EVALUATION

5.1. Benchmarks

To evaluate the effect of adding our prefetcher to the LEAP Scratchpads memory hierarchy, we examine a diverse set of previously published FPGA implementations. All codes we evaluate were originally expressed in terms of the generic request-response memory interfaces, and adding our prefetcher requires no modification. These codes are also several years old and were originally written targeting much smaller FPGAs than our evaluation platforms. As a result, adding our prefetching scheme does not materially impact the physical implementation of these designs in terms of area or maximum clock frequency. It is also important to note that these designs are highly performance-tuned: two of the designs [20, 21] won performance-based FPGA design contests.

Characteristic	**ML605**	**ACP**
Type	DRAM	SRAM
Capacity	512 MB	8 MB
Bandwidth	2780 MB/s	600 MB/s
Latency	254 ns	150 ns

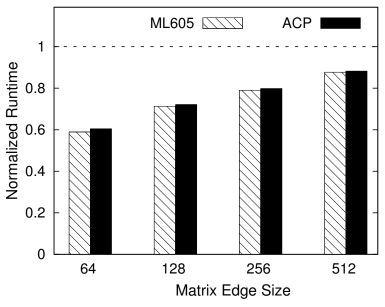
Table 1. Structural and performance metrics for evaluation FPGA platforms, as measured at the memory controller.

We arrange the benchmarks in order of the regularity of their memory access pattern.

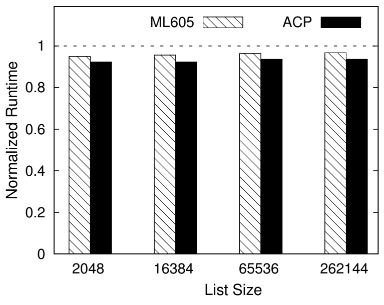
Blocked Matrix-matrix Multiplication: The MMM hardware [20] that we evaluate uses a block-style decomposition with its own internal block buffering. The two input and the output matrices are stored in separately initialized Scratchpad address spaces. When calculating a single output block, the hardware pulls in a complete row and column of the source matrices, computing a streaming multiply-accumulate. Although the block access pattern is non-strided, accesses within the rows of a single block, which are visible to our prefetcher, are strided. The fetch pattern of MMM is fixed statically and does not vary with input.

Merge Sort: Our merge sort implementation [21] achieves area-efficiency by time-multiplexing a single comparator among many lists. Lists are stored locally in FPGA SRAM, which is shared among all the lists to be merged. Multiple mergers may be concatenated to form a deep sorting pipeline, further reducing memory pressure. Merge sort uses a single Scratchpad interface. At runtime, the merger hardware repetitively picks a pair of lists and performs a single merge on them, propagating the result to the next merger. The item chosen at each list merge is data-dependent, causing the stored lists to drain at different and unpredictable rates. The hardware attempts to deal with this by observing the amount of data remaining in each list and fetching new data as the list drains. Thus, while the set of fetches produced by the merge sorter are determined statically, the order in which these fetches are issued by the hardware is data dependent.

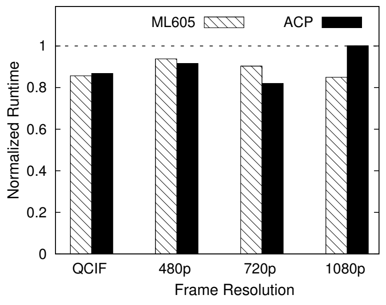
H.264: H.264 is a state-of-the-art video decoder. Like many video decoders, H.264 constructs new frames from portions of previously decoded frames stored in memory, a process known as inter-prediction. Pre-existing frames are generally



(a) MMM



(b) Merge Sort



(c) H.264

Fig. 3. Prefetcher performance results. Performance is normalized to a non-prefetched implementation.

fetched into the processing core in a raster-like order, left-to-right and top-to-bottom. However, there are significant sources of non-determinism in the fetch pattern. First, portions of the frame may not be inter-predicted, creating gaps in the memory access stream. Second, because there are several possible prediction modes, the memory access pattern may vary in stride at a fine grain. Our H.264 implementation [7] uses three separate Scratchpads, one for the luminance field and two for the chrominance fields.

5.2. Platforms

Because the evaluation benchmarks were written on top of LEAP, we can port implementations among different FPGA platforms without source modification by simply re-targeting the LEAP compilation flow to the new platform. Re-targeting a program does not change the functional behavior of a program, just as recompiling a C code for a new machine does not change the functional behavior of the C code.

In this study, we run benchmarks on two different FPGA memory hierarchies: the Nallatech ACP [22] module and the Xilinx ML605 [23]. The chief qualitative difference between the ACP and the ML605 is that the ACP board-level memory is a small, but fully-pipelined SRAM, while the ML605 board-level memory is a large DRAM, in which only some memory requests may be pipelined. Table 1 summarizes the properties of the memory systems of the two boards.

6. RESULTS

In this section, we first discuss our prefetcher's runtime performance results on each benchmark described in Section 5. Then, we use MMM as an example to analyze the prefetching accuracy and the effect of controlling prefetch bandwidth. We also discuss the area of our prefetching hardware. If not specified, we use the following configurations throughout this section: (1) each prefetcher has 32 learners; (2) look-ahead distance is increased with upper bound of 32 and decreased on every 4 timely prefetch hits; (3) prefetch requests are ignored when the total number of outstanding memory requests exceeds 12.

6.1. Runtime Acceleration

Blocked Matrix-matrix Multiplication: We evaluate our prefetcher architecture using matrix multiplications of various sizes. For small matrix sizes, the prefetcher provides up to a 40% performance gain.

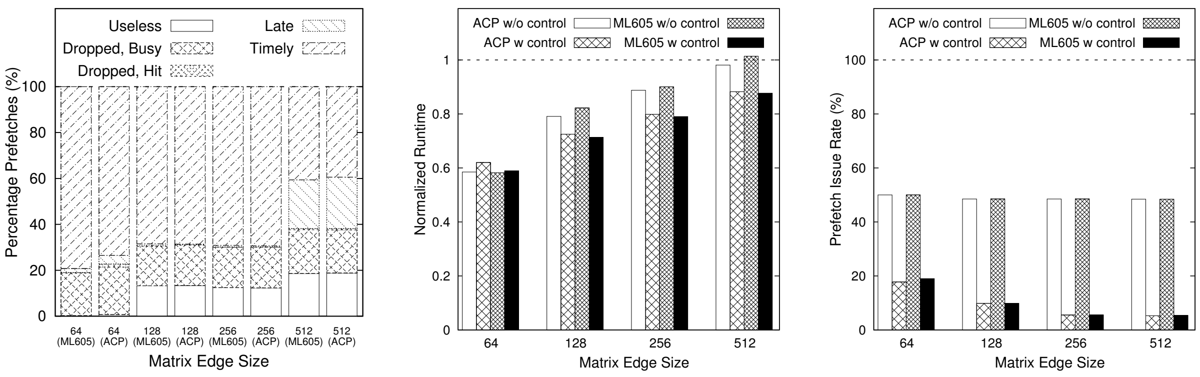
The underlying MMM hardware decouples memory operations from computation and actively attempts to overlap the latency of block loads with ongoing computation. When there are a sufficiently large number of blocks in a row or column, the hardware matrix multiplication hides most of the memory latency.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

Merge Sort: Merge sort is evaluated by sorting random lists of 128-bit integer values. Figure 3(b) shows that, our prefetching scheme marginally improves the performance of the merge sorter on both FPGAs.

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

H.264: Figure 3(c) shows the results of applying our prefetching scheme to an H.264 decoder operating on streams of various resolutions. Although the effect of prefetching varies among different video streams on different platforms, all benchmarks benefit from prefetching to some degree.

There are many factors that may cause the prefetching



(a) Prefetch Classification (b) Prefetch Runtime Performance (c) Prefetch Issue Rate

Fig. 4. Matrix-matrix multiplication result analysis

performance variation. First, because the encoding of a video is strongly dependent on the content of that video, the amount and kind of prefetching varies between encoded streams. Actually, for streams without inter-prediction, prefetching has no effect on performance.

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

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

6.2. Prefetch Accuracy

Figure 4(a) gives a classification of prefetches based on their dynamic behavior. Prefetches issued by our prefetcher microarchitecture fall into five categories. At issue, a prefetch may be dropped, either because the data already resides in the cache, or because the cache line target is busy, indicating that the prefetch is too late where the requested address may have been issued earlier by the client. Once issued, each prefetch has one of three possible outcomes: (1) timely useful, (2) late useful where the user program has already attempted to access the prefetch data, and (3) useless where the prefetch data is not accessed before it is evicted from the cache.

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

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

6.3. Prefetch Bandwidth Control

To prevent prefetching from overwhelming the memory bandwidth, our prefetcher automatically stops issuing requests when the number of outstanding memory requests exceeds a certain threshold, in effect permitting prefetching only if there is spare memory bandwidth available. Figure 4(b) and 4(c) show the effects of controlling prefetch bandwidth on runtime performance and prefetch issue rate. The low issue rate under bandwidth control indicates that MMM has high demand on the memory bandwidth, especially at large matrix sizes due to its own internal block prefetching. Without the bandwidth control, prefetch requests may overwhelm the memory and starve these memory requests, resulting in increased miss latency and runtime performance degradation. Bandwidth-limiting has a more pronounced effect on the ML605 because the DRAM is not fully pipelined.

6.4. Prefetcher Area

Table 2 shows the area of various implementations of our prefetching hardware. In general the area requirements of prefetching in the FPGA are extremely small, even if there are many stream learners incorporated into the prefetch engine. This is because the learner state, the chief consumer of area in an ASIC prefetcher, can be mapped efficiently to SRAM, leaving only the much smaller tracking and address generation logic to be implemented in slice resources.

	Slice Registers	**Slice LUTS**	**BRAM**	f_{max}
32 Learners, LUTRAM	333	1045	0	127 MHz
32 Learners, BRAM	419	1275	2	131 MHz
H.264, Baseline Profile	60770	86364	99	80 MHz

Table 2. FPGA resource utilization for prefetching logic. The area used by a single prefetcher is less than .5% of the total area of the LX240T chip used on the ML605 board. For comparison, the total area of our H.264 implementation is also shown.

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

7. CONCLUSION

FPGAs are establishing themselves as platforms for algorithm acceleration. However, many algorithm implementations do not fully utilize the resources available on the FPGA. Programs expressed using the Scratchpad memory abstraction leave the choice of memory system implementation to the compiler, permitting the compiler to leverage these unused resources. In this paper we have demonstrated that adding prefetchers to this synthesized memory hierarchy improves performance of pre-existing programs. The size of prefetchers is tunable to leverage excess resources, allowing performance tuning without design changes. In particular, inserting small stride prefetchers results in 15% average performance gain.

Although we have assumed in this paper the monolithic addition of prefetching as the sole means of memory system optimization, there are many possible optimizations to the memory system: improved caching policy, larger caches, and greater associativity. Such optimizations could be automated, with synthesis and behavioral feedback from a specific program dictating the optimizations applied by the compiler.

8. REFERENCES

[1] H. K.-H. So and R. Brodersen, "Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support," in *FPL*, 2006.

[2] M. Pellauer, M. Adler, D. Chiou, and J. Emer, "Soft Connections: Addressing the Hardware-Design Modularity Problem," in *DAC*, 2009.

[3] K. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. S. Emer, "Leveraging Latency-insensitivity to Ease Multiple FPGA Design," in *FPGA*, 2012.

[4] 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.

[5] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing," in *FPGA*, 2011.

[6] Xilinx, Inc., "Xilinx Memory Interface Generator," 2012. [Online]. Available: http://www.xilinx.com/

[7] K. Fleming, C.-C. Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks, "H.264 Decoder: A Case Study in Multiple Design Points," in *MEMOCODE*, 2008.

[8] A. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[9] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA*, 1990.

[10] S. Vanderwiel and D. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[11] J. Fu *et al.*, "Stride directed prefetching in scalar processors," in *MICRO*, 1992.

[12] K. Farkas *et al.*, "Memory-system design considerations for dynamically-scheduled processors," in *ISCA*, 1997.

[13] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ISCA*, 1994.

[14] S. Iacobovici *et al.*, "Effective stream-based and execution-based data prefetching," in *ICS*, 2004.

[15] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *ACM SIGARCH Comp. Arch. News*, vol. 25, no. 2, pp. 252–263, 1997.

[16] G. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: an application-driven study," in *ISCA*, 2002.

[17] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Trans. on*, vol. 44, no. 5, pp. 609–623, 1995.

[18] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," in *HPCA*, 2004.

[19] P. Diaz and M. Cintra, "Stream chaining: Exploiting multiple levels of correlation in data prefetching," in *ISCA*, 2009.

[20] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan, "Hardware acceleration of matrix multiplication on a xilinx fpga," in *MEMOCODE*, 2007.

[21] K. Fleming, M. King, M. C. Ng, A. Khan, and M. Vijayaraghavan, "High-throughput Pipelined Mergesort," in *MEMOCODE*, 2008.

[22] http://www.nallatech.com, "Nallatech ACP module."

[23] ——, "Xilinx ML605."