Improving Memory System Performance for Soft Vector Processors

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road
Toronto, Canada
{yiannac,steffan,jayar}@eecg.utoronto.ca

ABSTRACT

Recently proposed vector processing extensions [9, 10] can significantly improve the performance of a conventional FPGA-based soft processor, but significantly increase the pressure on the memory system to keep pace. In this work we investigate methods of improving the memory system for soft vector processors via (i) tuning the data cache configuration, namely its depth and line size, and (ii) hardware prefetching mechanisms. We evaluate on our VESPA soft vector processor connected to DDR and executing hand-vectorized benchmarks from the EEMBC industry-standard benchmark suite. We find that cache configuration provides a significant area/performance trade-off for designers to wield, providing near 2x average performance for 1.8x the system area. We also demonstrate that proper prefetching can improve performance by 28% on average, and up to 2.2x in the best case.

1. INTRODUCTION

Recent work has proposed extending soft processors with vector processing capabilities [9, 10] as a means of scaling performance for data-parallel workloads. Vector processing allows a single instruction to command multiple processor datapaths, or *vector lanes*, a parameter the designer can use to trade area and performance depending on the application and design constraints. However, while adding more vector lanes greatly increases the *compute* power of the soft vector processor, it simultaneously increases the pressure on the memory system to supply data at a proportionally fast rate. Thus, the design of a high-performance FPGA-based memory system is key.

Typical processor memory systems are cache-based, with many configurable parameters such as: i) levels of cache hierarchy, ii) associativity, iii) cache depth (the capacity of the cache), iv) cache line size, and v) prefetching. However, for FPGA-based processors the first two possibilities are less compelling. First, slower clock speeds of FPGAs result in a much narrower gap between soft processor performance and modern DRAM (in our case only 10 cycles), negating the benefits of multiple levels of cache. Second, associative caches are inefficient in FPGAs because of the relatively expensive multiplexing involved. Hence in this work we focus cache depth, line size, and prefetching.

Examining the operation of VESPA [9], our FPGA-based soft vector processor, we observed that on average **two-thirds of all processor cycles are spent waiting on the vector memory unit**. The vector memory unit causes such stalls because it is either i) accessing the data cache (often across many cache lines), or ii) bringing data into the data cache by reading from latent DDR SDRAM. In this work, we improve both of these cases by adding support for increasing data cache line sizes and prefetching data. Due to spatial locality, larger cache lines should increase the amount of useful data found in a single cache line hence reducing the number of cache accesses required to complete a vector memory instruction. Also, since data-parallel code typically has predictable memory access patterns, we can automatically prefetch data into the cache before the application requests it, tolerating memory latency.

A key benefit of data prefetching is that it allows the processor to make greater use of the burst-mode transfers common in every modern DRAM technology. For a conventional sequential soft processor bursts are typically small, limited to the size of the cache line; however, with data prefetching we can leverage the massive bandwidth provided by burst-mode transfers by transferring much larger blocks of data. This is a natural fit to vector processors since a single vector memory instruction targets a large amount of data. We can pass this information to the prefetcher and have it load into the cache all required data in (ideally) one memory transaction.

1.1 Related Work

The most closely related work to ours is by Yu *et. al.* [10], who demonstrated the potential for vector processing as a simple-to-use and scalable accelerator for soft processors, potentially scaling better than Altera's `C2H` behavioral synthesis tool for three benchmark kernels. However, that work modelled an on-chip 1-cycle (latency) memory system, hence disregarding the effects and exploration of more realistic memory systems.

Many strategies for accurate data prefetching have been explored in the computer architecture community as summarized by Vanderwiel and Lilja [6]. All strategies aim to prefetch needed data while minimizing the amount of *cache pollution*, i.e., prefetching useless data and/or evicting useful data. In our work we bring these ideas to FPGA-based vector processors and evaluate them in real hardware.

Fu and Patel investigated prefetching particularly in the context of a vector processor [3]. They limited prefetching to vector memory instructions with strides less than or equal to cache line size and found that prefetching is useful for up to 32 cache blocks—our results are evaluated in real FPGA

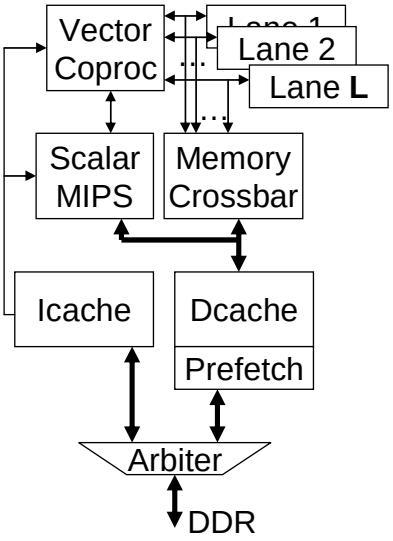

Figure 1: VESPA processor block diagram.

hardware with modern DRAM and agree with this. We further experiment with a *smart* vector prefetch where the vector length is used to calculate the number of cache lines to prefetch.

1.2 Contributions

In the context of a real hardware evaluation of memory systems connecting the VESPA soft vector processor to DDR SDRAM and running vectorized industry-standard EEMBC benchmarks, this paper makes the following two contributions: (ii) we quantify the large design and performance trade-offs in varying data cache depth and line size; (iii) we demonstrate that data prefetching can double performance for some applications.

2. VESPA

In our previous work on VESPA [9] we implemented a parameterized vector processor in hardware and explored its potential for scalability and customization. This paper extends VESPA by addressing the primary limit to scalability, namely the memory system. We further parameterize the data cache for VESPA to allow a designer to more powerfully trade area for performance scaling for data-parallel applications. In this section we provide a brief description of the VESPA architecture (see our previous paper for full details [9]).

The VESPA processor consists of a scalar MIPS processor which was automatically generated using the SPREE system [7, 8], coupled with a parameterized vector coprocessor based on the VIRAM [4] instruction set. Table 1 shows all the parameters for VESPA while Figure 1 shows a block diagram of the VESPA architecture. The scalar processor and vector coprocessor share the instruction stream and are both in-order pipelines, but can execute out-of-order with respect to each other except for memory operations which are serialized to maintain sequential consistency. Both share a direct-mapped data cache with parameterized aspect ratio (depth and cache line size). A crossbar connects each byte in a cache line to `M` of the `L` vector lanes in any given cycle. In this work we use only fully-connected memory crossbars by setting `M` equal to `L`.

Table 1: Configurable parameters for VESPA.

Parameter	Symbol	Values
Vector Lanes	L	1,2,4,8,16,. . .
Vector Lane Width	W	1,2,3,4,. . .
Maximum Vector Length	MVL	2,4,8,16,. . .
Memory Crossbar Lanes	M	1,2,4,8,. . . L
Each Vector Instruction	-	on/off
ICache Depth	ID	4KB,8KB,. . .
ICache Line Size	IW	16,32,64,. . .
DCache Depth	DD	4KB,8KB,. . .
DCache Line Size	DW	16,32,64,. . .
DCache Miss Prefetch	DPK	1,2,3,. . .
Vector Miss Prefetch	DPV	1,2,3,. . .

3. MEASUREMENT METHODOLOGY

In this section we describe the components of our infrastructure necessary to execute, verify, and evaluate VESPA. Specifically, we describe our hardware platform, verification process, CAD tool measurement methodology, benchmarks, and compiler.

Hardware Platform We use the multi-FPGA Transmogrifier 4 (TM4) [2] to host the complete vector processor systems. The platform has four Altera Stratix EP1S80F1508C6 devices each connected to two 1GB PC3200 CL3 DDR SDRAM DIMMs clocked at 133 MHz (266 MHz DDR). We use Altera's Quartus II 8.0 CAD software to synthesize our processor systems onto one of the four Stratix I FPGAs and clock the system at 50 MHz. Note our design was intended for a faster Stratix III FPGA where it has a clock speed of around 130 MHz, but we used the TM4 board since it was readily available. All instances of VESPA are fully tested in hardware using the built-in checksum values encoded into each EEMBC benchmark. Debugging is performed using Modelsim and is guided by comparing traces of all writes to the scalar and vector register files.

FPGA CAD Tools A key value of performing FPGA-based processor research directly on an FPGA is that we can attain high quality measurements of the area consumed and the clock frequency achieved—these are provided by the FPGA CAD tools. We use aggressive timing constraints to maximize the CAD tool's effort for default optimization settings but with register retiming and register duplication set to *on*. Through experimentation we found that these settings provided the best area, delay, and runtime trade-off. We also performed 8 such runs for every vector configuration to average out the non-determinism in modern CAD algorithms. The relative silicon area of each FPGA resource relative to a single logic element (LE) was supplied to us by Altera [1], and we used these equivalent areas to calculate the total silicon area consumed on the Stratix 1S80 measured in units of *equivalent LEs*—the silicon area of a single LE including its routing.

Benchmarks As listed in Table 2, for this study we use six benchmarks from the Telecom and Digital Entertainment suites of EEMBC, the industry-standard benchmark collec-

Table 2: EEMBC and Other Benchmarks Used.

Benchmark	Suite	EEMBC Dataset
AUTCOR	Telecom	2
CONVEN	Telecom	1
FBITAL	Telecom	2
VITERB	Telecom	2
RGBCMYK	Digital Ent.	5
RGBYIQ	Digital Ent.	6
IP_CHECKSUM	Networking	extract
IMGBLEND	Handmade	-
FILT3X3	Handmade	-

tion. We execute the largest dataset for each benchmark with the test harness and benchmarks uncompromised, potentially allowing us to calculate and report official EEMBC scores. We also use two hand-made benchmarks and one kernel extracted from the Networking suite in EEMBC; the datasets for these three benchmarks are hand made. Note that cycle counts are collected from a complete execution on our hardware platform as described above.

Compilation Framework Benchmarks are built using a MIPS port of GNU `gcc` 4.2.0 with `-O3` optimization level. Initial experiments with this version of `gcc`'s auto-vectorization capability showed that it is in its infancy, preventing us from automatically generating vectorized code from key EEMBC program loops. Instead we ported the GNU assembler to support VIRAM vector instructions and used hand-vectorized assembly EEMBC routines provided to us by Kozyrakis who used them during his work on the VIRAM processor [4].

VESPA Configuration In this paper we fix the configuration of VESPA to be a 16-lane vector processor with 64 element `MVL`, full 16-lane memory crossbar, full 32-bit datapath, and complete instruction set support. The instruction cache is configured as 4KB direct-mapped cache with 16B line size, and in general makes a negligible difference in performance because of the small loops typically executed in our embedded benchmarks. We study the data cache configuration in the next section.

4. CACHE DESIGN TRADE-OFFS

In this section we explore the speed/area trade-off for different data cache configurations. We vary data cache depth from 4KB to 64KB and the cache line size from 16 bytes to 128 bytes. We do not explore cache line sizes less than 16 bytes since the DDR memory transmits blocks of 16 bytes at a time so cache lines smaller than this would waste memory bandwidth. Similarly, we do not explore depths less than 4KB since a 16-byte wide cache would waste FPGA RAM storage bits if configured with less than 4KB of space (for the Stratix I) because of the discrete aspect ratios of the block RAMs. Finally we do not explore line sizes greater than 128 bytes or depths greater than 64KB because of various limitations of the CAD tools, device resources, and clock frequency.

4.1 Cache Line Size

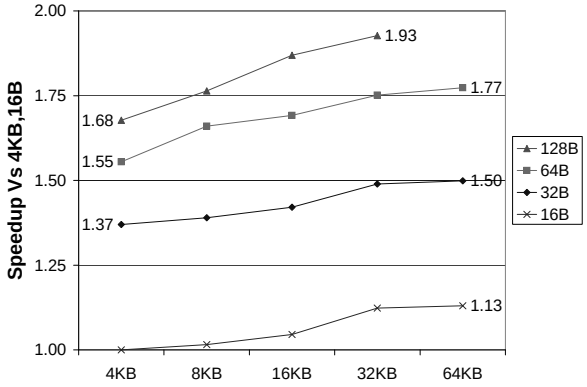

Figure 2: Wall clock speedup attained for different cache depths and cache line sizes over the 4KB cache with 16B line size. Each line in the graph depicts a different cache line size.

Figure 2 shows the average speedup across our benchmarks achieved for each data cache configuration normalized against the 4KB cache with 16B line size. We note first that the largest speedup achieved is almost 2x providing a large design space for area/speed trade-offs. The boost in computational power in the vector processor allowed the memory system to become more influential in determining overall performance, whereas previous work limited the role of memory in a (scalar) soft processor to 12% [5]. The figure also shows that cache line size is a much more dominant parameter than cache depth. This is largely due to the ability to satisfy more memory requests in the vector memory unit, but also due to the "prefetching" inherent in longer cache lines. Finally note that performance for a 64KB cache with 128B line size could not be attained because of the timing violations in our 50 MHz Stratix I design.

Although we use the Stratix I available to us for hardware evaluation, we measure clock frequency on a Stratic III 3S200C2 device since, as mentioned previously, VESPA was designed for that FPGA family. The clock frequency is slightly reduced as the cache line size increases. We found the frequency to be 129 MHz, 126 MHz, 123 MHz, and 122 MHz for 16B, 32B, 64B, 128B cache lines respectively. The frequency degradation is due to the multiplexing needed to get data words out of the large cache lines. Further logic design effort through pipelining and retiming can mitigate these effects resulting in slightly more pronounced benefits for the longer cache lines.

Figure 3 shows the silicon area of the complete system normalized against that of the 4KB cache with 16B line size. The area cost can be quite significant, in the worst case almost doubling the system area, but the area trends are quite different than what one would expect with traditional hard processors. We notice that increases in cache line sizes result in significant increases in area. This is largely due to bigger multiplexers, primarily in the vector memory unit crossbar which routes each byte to each vector lane; however it is also due to the increase in FPGA block RAMs being used. In their current configuration, the block RAMs are limited to only 16-bit wide data ports, therefore to create a cache with 16B (128 bit) line sizes we use 8 0.5KB *M4K* FPGA block RAMs in parallel, as shown in Figure 4, hence consuming all 8 of those block RAMs and their associated

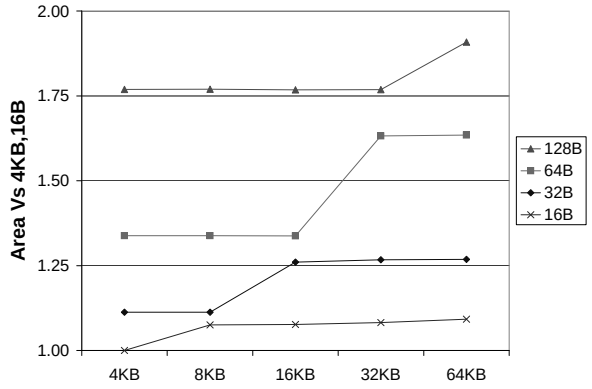

Figure 3: System area for different cache depths and cache line sizes normalized against the system with 4KB cache with 16B line size. Each line in the graph depicts a different cache line size.

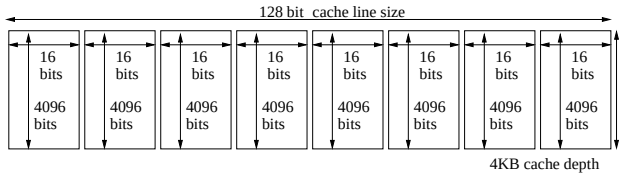

Figure 4: Multiple block RAMs are needed to create the width necessary for 16 byte cache lines. The cache depth should be configured as 4KB to fully utilize the capacity of the used block RAMs.

silicon area. Any increases in cache line sizes will result in corresponding increases in the number of used block RAMs and with it an automatic increase of physical storage used (whether it is logically used by the design or not) and silicon area consumed.

4.2 Cache Depth

Cache depth has a less pronounced impact on performance. It generally does not affect clock frequency, except for the 64KB, 128B cache where it violates timing. The streaming nature of many of the benchmarks minimize the need to store large working sets. In fact the performance gain begins to plateau after 32KB for all cache line sizes.

As shown in figure 3, growing the cache depth results in step-wise growth of the area as the cache gets implemented in a larger family of FPGA RAM blocks. When the cache depth exceeds the capacity of the number of block RAMs used to achieve the cache line size, the CAD tool uses the larger *MRAM* block RAMs on the Stratix instead. These MRAMs are both wider, up to 64 bits wide in their current configuration, and deeper, a total of 64KB. Using the MRAMs result in large spikes in area because of their massive size. Note that if MRAMs were not used, we would expect to see a more linear area growth with increased cache depth as more M4K block RAMs are used to achieve the desired depth.

5. IMPACT OF DATA PREFETCHING

A key advantage of our data prefetcher is that it leverages the high bandwidth from burst mode transfers, meaning

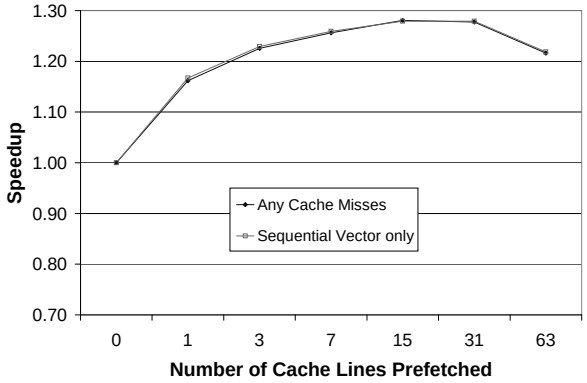

Figure 5: Average speedup, relative to no prefetching, of prefetching a varying number of cache lines using two strategies: (i) on *any data cache miss*; (ii) on any miss from a sequentially-accessing vector memory instruction.

after an initial miss penalty, all cache lines including the prefetched lines are streamed into the cache at the full DDR rate. This bandwidth is vital for VESPA which processes a batches of memory requests in each vector memory instruction. Complications arise from handling such large memory transfers when the evicted cache lines are dirty. To ensure these dirty lines are written to memory we need to either deny the prefetched line entry into the cache or buffer the dirty cache lines and later write them to memory, in our work we do the latter, but prefetching is halted when it reaches the end of the DRAM row that the miss initially accessed due to limitations in our design.

The data prefetcher is configured using the parameters `DPK` and `DPV` from Table 1. `DPK` is the number of consecutive cache lines prefetched on any cache miss—note that prefetching is triggered for both scalar and vector instructions since both share the same data cache and its prefetcher. To minimize cache pollution we introduce another parameter, `DPV`, to prefetch only on vector instructions with strides within two cache lines and are hence known to access the cache sequentially and can therefore be prefetched more aggressively.

Using the same base vector architecture from the previous section, we configure the data cache with 16KB depth and 64 byte line size and explore the effect of different data prefetching configurations. Because of limitations in our design we can currently only prefetch for 64 byte line sizes[1], we choose 16KB depth to fully utilize the used block RAMs. The aggressiveness of the prefetcher is measured by the number of cache lines it speculatively loads and is varied from 0 (off) to 63. As discussed earlier, we can separately configure the number of cache lines to prefetch for known sequentially accessing vector memory instructions. We explore both strategies in terms of benchmark performance and FPGA area, we do not discuss clock frequency since it was not affected by the prefetcher.

Figure 5 shows the performance, normalized against no prefetching, of prefetching on any miss and prefetching only for sequentially-accessing vector instructions. The figure

[1]Significant modifications would be needed for our DDR controller to support prefetching for smaller cache line sizes, while the 128 byte line size is unstable in hardware.

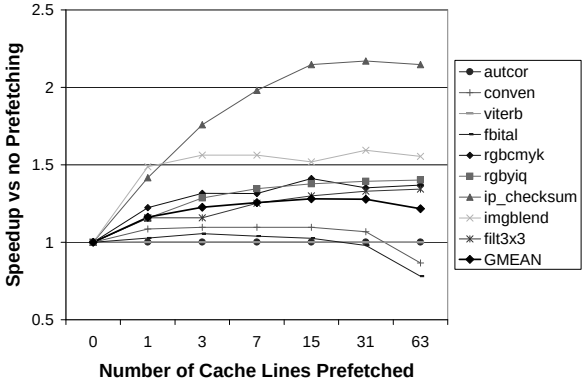

Figure 6: Speedup when prefetching a constant number of consecutive cache lines on any data cache miss, relative to no-prefetching.

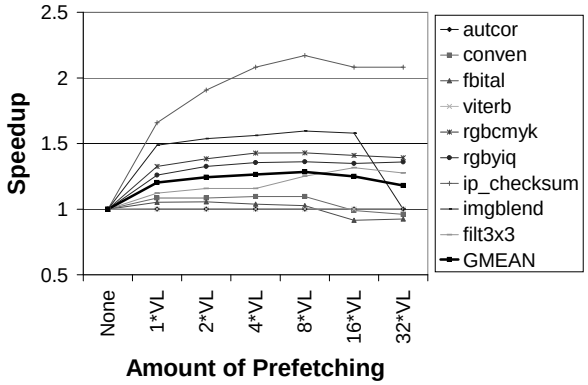

Figure 7: Speedup when prefetching a constant number of consecutive cache lines on any data cache miss, relative to no-prefetching.

shows that the performance is the same—they are collinear in the graph. This is partly expected since our benchmarks generally use very few scalar loads or stores and generally have vector strides less than a cache line, thus most memory operations come from sequential vector instructions. The speedup achieved is quite significant resulting on average 28% faster performance. As we increase the number of cache lines prefetched, and hence the aggressiveness of the prefetcher, we see diminishing returns on the performance gains until the cache pollution dominates, reducing the speedup at 31 and 63 cache line prefetches.

Figure 6 shows the performance of each benchmark as the aggressiveness is increased for prefetches triggered by any cache miss. The graph shows that four of the benchmarks, AUTCOR, CONVEN, FBITAL, and VITERB do not benefit significantly from prefetching, while the other five benchmarks achieve large speedups as high as 2.2x. For large prefetches the performance tapers off and then begins a downward trend, in the case of CONVEN and FBITAL the performance becomes worse than with no prefetching. As long as the number of cache lines being prefetched is moderate, benchmarks which benefit from prefetching can achieve large speedups without slowing down benchmarks which do not. Of course the benefit of using a soft vector processor is that one can tune the amount of prefetching for each application. For example, 15 is on average the best, but IMGBLEND is best served by prefetching 31 cache lines while 15 performs worse than most other configurations

With respect to area, the cost of prefetching is largely dominated by the writeback buffer which stores dirty cache lines that have been evicted. In general the buffer needs to have the greater of `DPK`+1 or `DPV`+1 entries for the case where all evicted lines are dirty. With prefetching disabled this cost is reduced to a single register, but otherwise is generally implemented in FPGA block RAMs where the effect of discrete aspect ratios as discussed in figure 4 result in a constant cost when prefetching is between 1 and 15 cache lines amounting to only 1.6% of total system area. For more than 15 cache lines this area cost doubles, but as mentioned previously, there is no additional performance gain seen in our benchmarks making the trade-off uninteresting.

5.1 Vector Length Prefetching

Choosing an optimal value for `DPV` depends on the mix of vector lengths used in the program, but each vector memory instruction instance explicitly specifies its vector length providing a valuable hint for `DPV`. We therefore enable a *vector length* prefetching mode which prefetches all data required by a vector memory instruction once that instruction experiences a cache miss. Note that the actual vector length used is the remaining vector length, so if the first eight elements of a vector load were cache hits and a miss occurred on the ninth element of a 64 element vector, the prefetcher would use 55 as the vector length.

Figure 7 shows the performance of a range of vector length prefetches. Prefetching $8VL$ cache lines performs best achieving a maximum speedup of 2.2x for IP_CHECKSUM and 29% on average. Of specific interest is the $1VL$ configuration which prefetches the remaining elements in a vector miss and hence has zero cache pollution. This configuration has no speculation, it guarantees no more than one miss per vector memory instruction and is ideal for heavily mixed scalar/vector applications, but only achieves 21% speedup on our benchmarks. This shows that our benchmarks benefit from prefetching cache lines needed on subsequent loop iterations, but too much prefetching can undo the performance gains as seen in IMGBLEND where large prefetches of the input stream conflicts in the cache with the output stream causing thrashing between the two streams.

There is a slight additional area cost of 0.3% for the vector length prefetcher which needs to compute the number of cache lines to prefetch. This computation includes a multiply operation making the area cost non-negligible.

5.2 Cache Line Size and Prefetching

It is important to realize that cache lines already perform some intrinsic prefetching: the larger the cache line, the more data that is effectively prefetched, likely reducing the benefit of additional hardware prefetching. Since we can only implement prefetching for 64B cache lines, we are unable to evaluate the relationship between the two in hardware, but we expect results similar to Fu and Patel [3]. One key result from this work is that increasing cache line size has far higher area costs than prefetching due to the growth of the memory crossbar and increase in FPGA block RAMs required. However longer cache line sizes provide additional performance benefits in terms of higher cache-to-

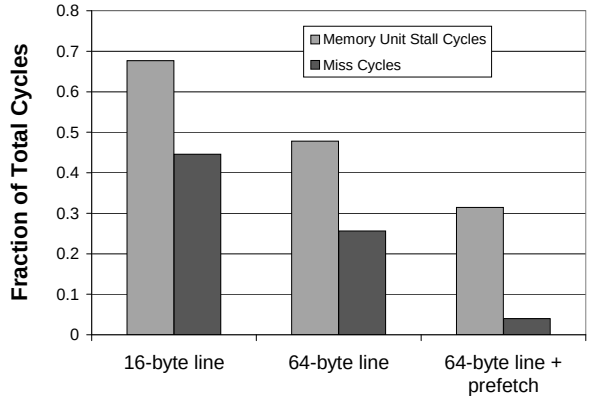

Figure 8: Average fraction of simulated cycles for AUTCOR, CONVEN, IP_CHECKSUM, and IMGBLEND spent waiting in the vector memory unit in total or exclusively servicing a miss for a 16-lane full memory crossbar VESPA processor when cache lines are widened to 64B and prefetching is enabled for the next 15 cache lines.

lane throughput whereas prefetching helps only memory-to-cache throughput. We advocate an approach where cache line size is maximized subject to area constraints, and prefetching is used to further reduce the occurence of misses. The effect of this approach in our own work is discussed below.

Figure 8 shows the average number of cycles across four of our benchmarks spent waiting on the vector memory unit or specifically waiting for missed data to be retrieved from memory. The figure demonstrates that our original 16-lane full memory crossbar version of VESPA with 16B data cache lines and 4KB depth spends two out of three cycles waiting on the vector memory unit, suggesting that memory system performance is throttling VESPA's performance. Expanding the cache line (and the depth to match the block RAM usage) reduces the cycles spent waiting on the vector memory unit to less than half. Finally by adding prefetching we reduce it to only one out of three cycles and we note that miss cycles were reduced to just 4%, leaving little motivation to further improving the prefetching.

6. CONCLUSIONS

Vector coprocessors provide large gains in soft processor computing power but necessitate a corresponding gain in memory system performance to keep pace. Tuning the data cache size and line size is an effective means of trading area for performance in our VESPA processor system, and provides approximately a doubling of performance for a system with double the silicon area. Cache line size was shown to be a more dominant parameter in determining both area and performance, while cache depth is largely discretized by the nature of the FPGA block RAMs.

Data prefetching is a very effective means of: (i) parallelizing memory transfers alongisde computation; (ii) leveraging the large bandwidths available from burst-mode transfers in modern DRAMs; and (iii) utilizing the memory access pattern information encoded into vector memory instructions. In this work we explored sequential prefetching of a contant number of cache lines for all cache misses or those from sequentially accessing vector instructions. We also enable a vector length prefetcher which dynamically calculates the number of cache lines to prefetch from the vector length. Large prefetches of 8 times the vector length achieved the best overall performance with 29% speedup, while a conservative vector length prefetcher achieves 21% speedup and avoids the need to tune the the number of lines to prefetch to the application.

In future work we plan to investigate ways to further improve the execution of memory instructions. We believe there are opportunities for smart banking configurations in the data cache allowing single-cycle access for vectors that span or stride across multiple cache lines. We are also in the process of migrating our hardware platform to the Altera DE3 board which uses state-of-the art Stratix III FPGAs. This board will allow further vector lane scaling and will hence further require better memory system performance.

7. REFERENCES

[1] R. Cliff. Altera Corporation. Private Comm, 2005.

[2] J. Fender, J. Rose, and D. R. Galloway. The transmogrifier-4: An fpga-based hardware development system with multi-gigabyte memory capacity and high host and memory bandwidth. In *IEEE International Conference on Field Programmable Technology*, pages 301–302, 2005.

[3] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. *SIGARCH Comput. Archit. News*, 19(3):54–63, 1991.

[4] C. Kozyrakis and D. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36–45, 2003.

[5] M. Labrecque, P. Yiannacouras, and J. G. Steffan. Scaling Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'08).*, Palo Alto, CA, April 2008.

[6] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[7] P. Yiannacouras, J. Rose, and J. G. Steffan. The Microarchitecture of FPGA Based Soft Processors. In *CASES'05: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 202–212. ACM Press, 2005.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose. Application-specific customization of soft processor microarchitecture. In *FPGA'06: Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 201–210, New York, NY, USA, 2006. ACM Press.

[9] P. Yiannacouras, J. G. Steffan, and J. Rose. Vespa: Portable, scalable, and flexible fpga-based vector processors. In *CASES'08: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2008.

[10] J. Yu, G. Lemieux, and C. Eagleston. Vector processing as a soft-core cpu accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232, New York, NY, USA, 2008. ACM.