TPUTCACHE: HIGH-FREQUENCY, MULTI-WAY CACHE FOR HIGH-THROUGHPUT FPGA APPLICATIONS

Aaron Severance, Guy G.F. Lemieux

Department of Electrical and Computer Engineering
University of British Columbia

email: aaronsev@ece.ubc.ca, lemieux@ece.ubc.ca

ABSTRACT

Throughput processing involves using many different contexts or threads to solve multiple problems or subproblems in parallel, where the size of the problem is large enough that latency can be tolerated. Bandwidth is required to support multiple concurrent executions, however, and utilizing multiple external memory channels is costly. For small working sets, FPGA designers can use on-chip BRAMs achieve the necessary bandwidth without increasing the sytem cost. Designing algorithms around fixed-size local memories is difficult, however, as there is no graceful fallback if the problem size exceeds the amount of local memory. This paper introduces TputCache, a cache designed to meet the needs of throughput processing on FPGAs, giving the throughput performance of on-chip BRAMs when the problem size fits in local memory. The design utilizes a replay based architecture to achive high frequency with very low resource overheads.

1. INTRODUCTION

There is growing interest in using FPGAs to do data processing, harnessing the massive parallelism available and low power operation without having to create a custom ASIC. As well, FPGAs are used in embedded designs for connectivity due to their flexible I/Os; since data is passing through the FPGA anyway any data processing that can be done inside the FPGA potentially saves power and board space. FPGAs can support many types of parallelism, including bit-level, pipeline, and data parallelism. Data parallelism that is not latency sensitive lends itself to *throughput* processing, where multiple data elements are interleaved so each stage of execution is processed in a time-multiplexed manner. This allows for on-chip resources to be used mostly for computation rather than expensive control structures such as hazard detection and data forwarding. A flexible *overlay* architecture may implement throughput processing suitable for a variety of applications while supporting software-like algorithm compile and download times (seconds) compared to the long FPGA synthesis design cycle (tens of minutes to hours).

Of interest in supporting such overlays is the memory system, which must be flexible enough to support different applications without going through synthesis again. For simple applications, data for each context may be statically analyzable and fit into on-chip memory, in which case the FPGA's BRAMs can be configured to be a fixed-size global memory, as in Figure 1. If data is too large to fit on-chip external memory may be used, but going off-chip carries performance and power penalties. An alternative for applications with large or unknown size data sets is to create a cache out of memory-mapped BRAMs, as in Figure 2. Ideally, the cache would allow for the same level of performance as BRAMs when data can be mapped to the size of the cache. At the same time, it would function as a traditional cache, increasing performance in other situations as long as some locality of reference existed. Low resource overhead would be needed so that designers would be willing to tradeoff any increase in logic and BRAM usage rather than laboriously redesigning their algorithms to manually move data between on-chip and external memory.

While caches have been implemented in FPGAs many times before, they do not meet our need for throughput computing for the following reasons:

- F_{max} requirements: to be a viable BRAM replacement the cache should be able to operate at near BRAM frequencies. This will require deep pipelining; single-cycle latency will not be possible. In systems that can only run at a fraction of BRAM F_{max} multipumping can be used to provide additional ports.
- Fully pipelined: The design must be able to provide one hit per cycle on all ports when hitting in the cache to provide BRAM level throughput.
- Support multiple outstanding misses: Blocking on a

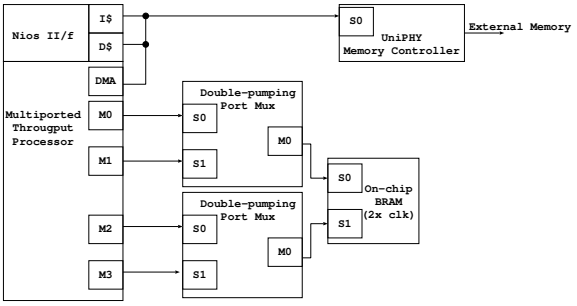

Fig. 1. System Design Utilizing BRAM As Shared Scratchpad

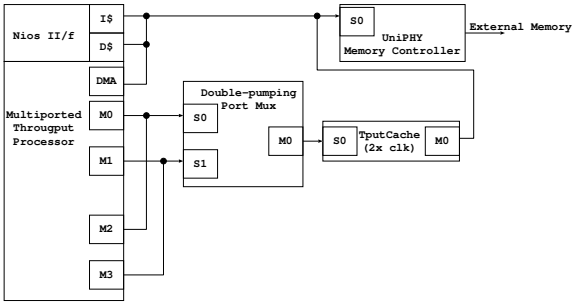

Fig. 2. System Using TputCache

miss will degrade performance on throughput applications, where some contexts may be able to make forward progress while others wait. This would also help if the cache was placed in front of a high latency external interface such as flash memory.

- Write coalescing: A throughput application may write many words on the same line before reading from that line, in which case write-through caches will not save external memory bandwidth. Writeback caches with an allocate on write miss policy are required at the least.

- Arbitrary associativity: To allow multiple contexts to coexist in the cache without undue effort on the part of the programmer to avoid cache indexing conflicts, high associativity should be supported. Previous work has looked at direct mapped and two-way set associative caches.

To address these concerns, this paper introduces TputCache. TputCache uses a simple replay-based architecture to achieved fully pipelined, high frequency operation without undue added latency. It achieves a frequency of up to 253 MHz in a Cyclone IV device using a six-stage pipeline, comprable to the 270 MHz maximum frequency of BRAMs. Though requests are serviced in-order, it can have multiple outstanding misses in its replay pipelining, allowing for prefetching up to six outstanding misses concurrently.

2. BACKGROUND AND RELATED WORK

Previous work on overlay architectures for throughput computing includes highly multithreaded processors [1] and soft vector processors (SVPs) [2]. These overlays allow a throughput-amenable program to be written in software and downloaded and executed without performing synthesis again. Multithreaded processors are programmed for throughput computing in a single-instruction multiple-thread (SIMT) fashion, where multiple copies of the same thread are used to process data in parallel. SVPs, by contrast, run a single thread of execution but operate on many data elements in parallel. These operations may be executed in parallel in a single-instruction multiple-data (SIMD) fashion as well as pipelined over multiple cycles. This paper uses MXP, an SVP that includes scatter-gather support, to exercise TputCache. Scatter-gather uses a vector of addresses to do either multiple stores (scatter) or loads (gather) concurrently. The acceses may therefore be correlated or random depending on the application's access patterns, and may be both parallel (SIMD) and pipelined.

Previous FPGA cache work includes Choi et al. [3], which focuses on implementing multiple port caches with single cycle access latency. Their investigation looks at both mutlipumped and live-value table approaches, and gives from 2 to 7 ports with single cycle latency at up to 138.9 MHz on a Stratix IV. However, optimizing for single cycle latency severely limits F_{max} and is not necessary in throughput computing, where multiple contexts are used to allow for latency tolerance. The blocking design of the cache also limits its usefulness in a throughput environment, where multiple cache misses may be outstanding concurrently. A similar work is the parameterized cache generator of Yiannacouras el al. [4], which is also not optimized for throughput.

Other works such as CHiMPS [5] focus on synthesizing multiple caches for a given application to give a large number of ports to aid in throughput. These designs are fixed at synthesis, though, and so are not suitable for fixed overlays. They also rely upon the programmer or compiler to restrict accesses to not overlap and flush data from caches at certain points, limiting their general purpose usefulness. Reconfigurable caches, as in Gil et al. [6] are a general purpose cache solution, but sacrifice absolute performance for runtime configurability.

3. TPUTCACHE DESIGN AND ARCHITECTURE

Figure 2 shows an example system with a multiported accelerator connected to TputCache. TputCache is currently has a single internal facing port, but can run at close to

>

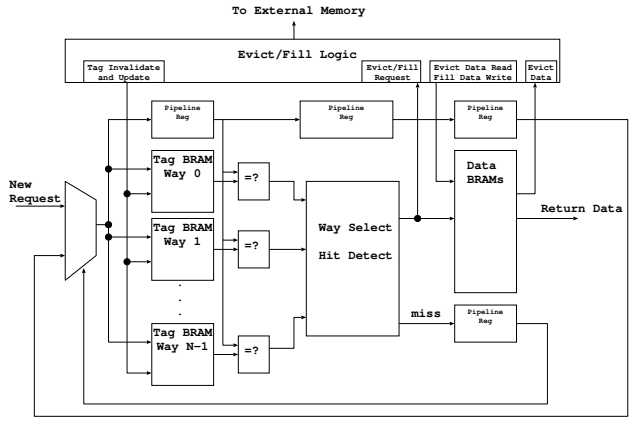

Fig. 3. TputCache Pipeline

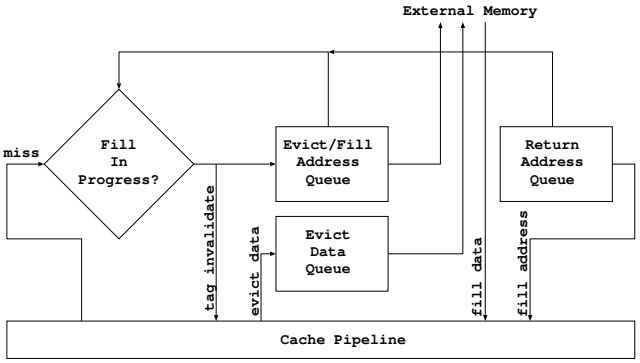

Fig. 4. Fill/Evict Pipeline

BRAM fmax. This contrasts with our tested accelerator, MXP, which has a memory interface that runs at 1/2 BRAM fmax (using multipumping internally for its BRAM based scratchpad). To achieve higher throughput than directly connecting MXP to TputCache, a doublepumping 2to1 mux is placed on TputCache's internal port.

TputCache is built in genericized VHDL, with top level generics for size, associativity, request width, external memory width, and line width. Figure 3 shows the replay based architecture. A six-stage pipeline is used, consisting of tag lookup, hit detection and way selection, and finally data ram write or read. Cache misses are dealt with by reinjecting missed requests and replaying them until the request finally hits; missed requests are sent to the fill pipeline that reads new lines from memory and evicts the lines' previous occpuants. By using a replay architecture no stalling is needed, and no multiplexers are needed after the initial mux to select a new request or replay.

Initially requests come in on the two input ports operating at half the cache frequency and are passed through the 2to1 doublepumping mux. Each request is tagged with a requestor signal so that requests to different ports can complete out of order with respect to each other. Each port's requests will complete in order, but it is expected that for throughput applications there will be no need to guarantee ordering between ports.

Once requests have been translated to TputCache's internal frequency, they enter the main pipeline. Stage 0 of the pipeline muxes in either a new request or a replayed request. If a new request is inserted into the pipeline, a token is also placed into a request number token FIFO which will be later used to ensure that requests complete in-order. The request address is then broadcast to a configurable number of ways in the cache, with a separate tag BRAM for each way. Because the pipeline operates at near the maximum BRAM frequency, output registers are used on the tag BRAM so tag lookup is not available until stage 2. On stage 2 each way uses a comparator to check for hit or miss. Stage 3 is used to aggregate the hit and miss signals from each way and select the correct way. Stage 4 drives the way and address to data BRAMs. At this point a write hit will commit if its request number matches the token FIFO head. A read hit requires two extra cycles before data is available (again, the output registers on the BRAM are used). On Stage 6 a read hit can commit if the return port is ready and the request number matches the token FIFO head.

If a request is not marked as committed, after Stage 6 it is reinserted into the beginning of the replay pipeline. This allows requests to be out-of-order within the pipeline, and is the reason for the token FIFO. The token FIFO is a simple way to ensure all requests commit in-order. More aggressive out-of-order retirement would be possible if required for performance but was not necessary for our test cases. Out-of-order reads (hit under miss) would require associative structures (which are expensive to implement in FPGAs) to reorder data before returning it to the system interconnect.

On a miss, the miss address and a way selected for replacement are passed into the fill pipeline, shown in Figure 4. Way replacement is random, fed by a 16-bit LFSR from which the appropriate number of bits are selected for the number of ways instantiated. If there are no misses to the same line pending, the fill pipeline places the address to fill, as well as the address of the evicted cache line are placed into the evict address queue. On the same cycle, the corresponding tag BRAM is invalidated so that no further writes to the line can occur. After the line is invalidated there may still be requests which have hit in the tag ram but are still in the pipeline, so a delay of four cycles is inserted to flush any requests to the evicted line.

Once the evict address queue has a valid entry (and data is in the evict data queue if necessary) memory requests are issued for the evict and the fill. The fill address and way are placed into a writeback queue similar to the evict address

queue. When data is returned from external memory the return address queue supplies the way and address of the tag BRAM to be updated at the same time the data BRAMs are written.

The address queues for both evict address and return are be used for hazard detection against new miss requests to ensure that a miss is not enqueued multiple times; this requires reading them out in parallel. Though this means they cannot be placed into FPGA memories (placed into FFs instead), only a small number of entries is needed to track outstanding misses (serving the function that MSHRs serve in a traditional lockup-free cache). For our test board, there was no increase in performance moving to more than 2 entries in each queue. Evict data is held in a separate queue and does not need to be read in parallel, so it is implemented with FPGA memories.

4. RESULTS

All soft processor results in this paper are measured by running the benchmarks on an Altera DE2-115 development system using Quartus II version 12.0sp2.

4.1. Area and Clock Frequency

Table 1 shows results from implementing various cache configurations on the Cyclone IV EP4CE115 used for benchmarking, as well as a Stratix IV EP4SGX530. On the Cyclone IV, F_{max} reaches up to 253 MHz, and decreases as the size and associativity of the cache increases. The design goal was to operate at the BRAM F_{max} of 270 MHz; critical paths in the token FIFO and fill logic have decreased F_{max} by 7% for the smallest implementation tested. Larger caches have lower F_{max} due to higher routing delays as more BRAMs must be used, increasing routing distance and congestion.

Logic usage ranges from 2632 to 3383 LEs including the port multiplexors and buffering. The cache itself takes 1709 to 2371 LEs, about the size of a Nios II/f processor (1810 LEs). One M9K is used per kB of data RAM, with tag RAMs using two to four BRAMs per way depending on the way depth, and an additional M9K used by evict data FIFO. This gives an overhead of 7% to 28% in BRAM usage.

Though TputCache was not optimized for Stratix devices Table 1 shows Stratix IV results for the same device used in Choi et al.[3]. TputCache achieves 422.8 MHz (211.4 MHz port clock) compared to their maximum RAM clock of 271.6 MHz (135.8 MHz port clock). Area numbers are only given for the whole system including accelerator and cache (13,460 ALMs), so a direct comparison is not possible. TputCache uses 2050 ALMs for the core; 2582 including the 2to1 doublepumping mux.

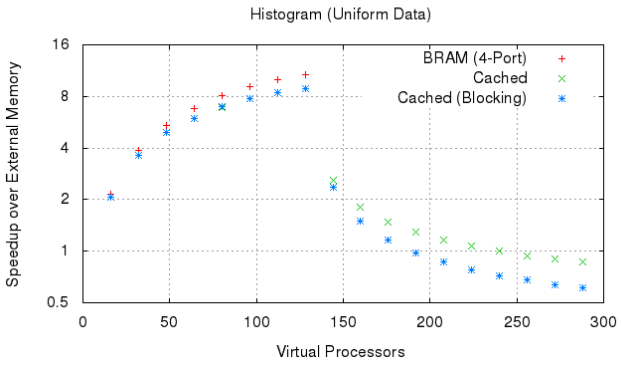

Fig. 5. Histogram of Uniformly Distributed Random Data

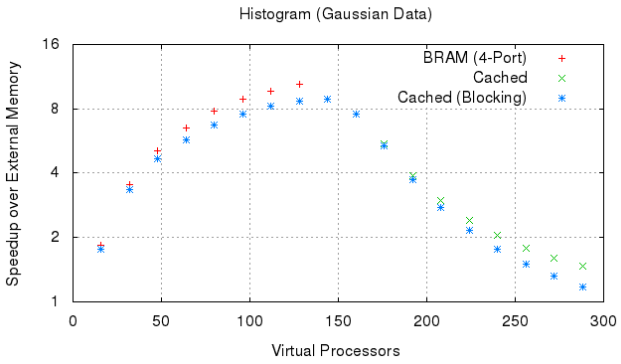

Fig. 6. Histogram of Gaussian Distributed Random Data

4.2. Soft Vector Processor Benchmarks

The following benchmarks are written for the MXP soft vector processor and use scatter-gather instructions for data dependent memory operations. MXP was configured with 16 vector lanes, 128kB of scratchpad memory, and 4 scatter-gather ports. The cache used in all benchmarks was configured as a 4-way, 128kB cache with 32-byte lines. For comparisons with memory mapped BRAMs, a 128kB double-pumped BRAM was used to give 4 ports compared to the cache's 2. Scatter-gather memory operations were mapped into cacheable memory space while other data was left uncached to avoid pollution. Additionally a 'blocking' version of the cache was created where only one evict-fill memory operation was allowed in flight at a time to demonstrate the value of prefetching multiple outstanding misses. Results are presented normalized to running the alogrithm from the

Table 1. Resource Usage (32B Line Size)

Parameters		Cyclone IV EP4CE115				Stratix IV EP4SGX530			
Ways	Size (kB)	M9Ks	F_{max} (MHz)	Cache LEs	LEs w/2x Adapter	M9Ks	F_{max} (MHz)	Cache ALMs	ALMs w/2x Adapter
4	32	41	253	1840	2726	41	423	2050	2582
4	64	73	244	1709	2632	73	398	2144	2636
4	128	141	231	1806	2683	137	385	2234	2718
4	256	273	223	1871	3023	265	333	2641	3157
8	256	281	213	1969	3084	273	322	2689	3178
16	256	289	213	2371	3383	289	340	2875	3390
Device Maximum		432	270	114480		1280	490	212480	

external memory of the DE2-115 board.

Figure 5 shows the results of histogramming 4M uniformly distributed randomly generated 32-bit numbers into 256 bins. The vectorized algorithm uses a number of virtual processors (VPs) which each histogram a subset of the data. Each VP requires a separate set of 256 counters, as incrementing a counter is a read-modify-write operation that is not atomic. With 32-bit counters, this means 1kB of state is needed for each VP. The VPs are time multiplexed on to physical lanes, so as the number of VPs is increased the latency of the scatter-gather operations is amortized over multiple sequential store-load operations and performance increases.

There is a considerable gain going from external memory to cache, 8.9x at 128 VPs. A double-pumped BRAM gives slightly higher performance, up to 10.5x at 128 VPs. After 128VPs, cache performance degrades until at 240 VPs, when the working set is 2x the size of the cache, performance becomes slightly worse than just using external memory.

Using uniformly distributed data does not exploit spatial locality in the cache and is the worst case; Figure 6 shows the results of histogramming a Gaussian distribution (approximated by adding multiple uniformly distributed data sets together and normalizing). When there is locality in the data, performance degrades much more gradually, and stays above that of external memory with a working set more than 2x the size of the cache.

Figure 7 shows the results of doing the Hough transform on different sized images. The Hough transform converts a greyscale image to a 2D plane representing the angle and distance from origin of a line; peaks in Hough space correspond to lines in the original image. For each pixel in the input image, it calculates the set of (angle, distance) pairs of lines intersecting it and adds the pixel value to those locations in the output image.

Speedups over external memory reach up to 2.7X for a 820kB output image. Speedup increases past the 128kB size of the cache as each input pixel will modify a subset of the output image, and nearby pixels in the input image will have close (angle, distance) pairs in the output. As the output image continues to grow larger eventually conflicts in the cache force out cache lines that could be used on the next pixel, leading to reduced performance, but since random way replacement is used performance degrades gracefully and stays above 1.8X that of external memory. Servicing multiple outstanding requests helps as the image size increases, giving up to an 11% boost over the blocking version of the cache.

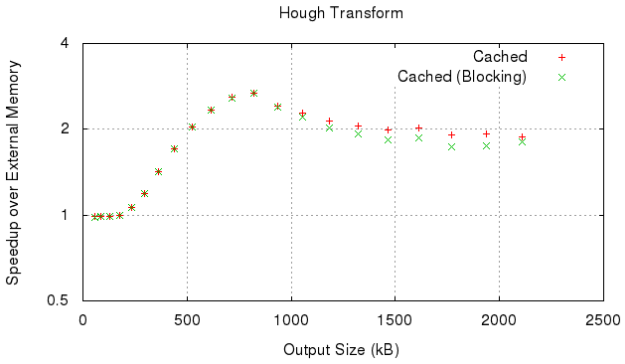

Fig. 7. Hough Transform

The motion compensation benchmark in Figure 8 uses a custom memory instruction attached to MXP to perform motion compensation according to the H.264 standard (only full-pel is supported currently, though). For this test, motion vectors corresponding to +/- 16 pixels with a Gaussian distribution (calculated in the same manner as the histogram benchmark) were used, with a different motion vector for every 4x4 pixel block.

Since the motion vectors are relatively small (local), which is typical of real video data, much of the performance increase comes from taking advantage of spatial locality. The size of the image has little effect on the speed of the algorithm; rather the increase in speedup over external memory as the image size grows larger comes from the external memory slowing down at larger image sizes. As the row size of the image increases, nearby row accesses become less likely to hit in the DRAM's row buffer and performance decreases; the cache amortizes this over multiple read hits.

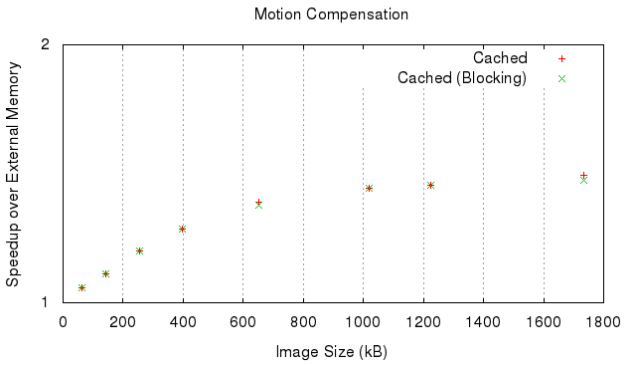

Fig. 8. Motion Compensation

5. FUTURE WORK

TputCache currently supports only one input port (two if run at 2x system clock frequency). Currently the dual ported tag and data BRAMs used have one port dedicated to the main pipeline and one to the fill logic. A second request port could be added by multiplexing between the fill logic and a request. This would give the same number of ports as simply using BRAMs, though needing additional multiplexing and control logic.

Other ways to increase ports inclue multiporting techniques such as those in LaForest & Steffan [7], but these incur resource and frequency penalties. Banking the cache based on address bits can also be used at the cost of stalling when there is a collision. TputCache, with its separate core pipeline and fill logic, could be banked fashion where multiple narrow cores are connected to a single wide (external memory burst width, 256 to 1024-bit), arbitrated fill logic module. Lines could therefore be spatially contiguous in memory, yet sequential accesses would hit in different banks.

For uses such as scatter operations or DMA writes that may store entire cache lines without reading them, it would be desirable to allow TputCache to act as a write cache. Instead of reading in a cacheline on a write miss, a write cache simply clears an existing line and tracks which bytes are dirty (and need to be written back). The BRAMs' 9th bit (currently unused) could track the dirty state of each byte of a line, increasing performance on long write operations by eliminating unnecessary cache line reads.

6. CONCLUSIONS

TputCache allows designers of throughput processing applications on FPGAs to utilize a cache memory with only a small loss of throughput compared to a static memory mapped BRAM. Reaching an F_{max} of 253 MHz, within 7% of that of the BRAM F_{max}, TputCache demonstrates that for latency tolerant applications on FPGAs there is a reasonable point of tradeoff between ease-of-use and performance. Future work will continue to expand the areas where TputCache can be used instead of multiple on-chip memories.

7. ACKNOWLEDGMENTS

The authors would like to thank NSERC, VectorBlox Computing, and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC for funding and Altera for donating development boards.

8. REFERENCES

[1] C. E. LaForest and J. G. Steffan, "Octavo: an fpga-centric processor family," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 219–228. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145731

[2] A. Severance and G. Lemieux, "Venice: A compact vector processor for fpga applications," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec., pp. 261–268.

[3] 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 *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 29 2012-May 1, pp. 17–24.

[4] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for fpgas," in *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, Dec., pp. 324–327.

[5] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, "Chimps: A c-level compilation flow for hybrid cpu-fpga architectures," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept., pp. 173–178.

[6] A. Gil, J. Benitez, M. Calvio, and E. Gomez, "Reconfigurable cache implemented on an fpga," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, Dec., pp. 250–255.

[7] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 41–50. [Online]. Available: http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/1723112.1723122