2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems

Jongsok Choi*, Kevin Nam*, Andrew Canis*, Jason Anderson*,
Stephen Brown*, and Tomasz Czajkowski†
*ECE Department, University of Toronto, Toronto, ON, Canada
†Altera Toronto Technology Centre, Toronto, ON, Canada

***Abstract*—We describe new multi-ported cache designs suitable for use in FPGA-based processor/parallel-accelerator systems, and evaluate their impact on application performance and area. The baseline system comprises a MIPS soft processor and custom hardware accelerators with a shared memory architecture: on-FPGA L1 cache backed by off-chip DDR2 SDRAM. Within this general system model, we evaluate traditional cache design parameters (cache size, line size, associativity). In the parallel accelerator context, we examine the impact of the cache design and its interface. Specifically, we look at how the number of cache ports affects performance when multiple hardware accelerators operate (and access memory) in parallel, and evaluate two different hardware implementations of multi-ported caches using: 1) multi-pumping, and 2) a recently-published approach based on the concept of a live-value table. Results show that application performance depends strongly on the cache interface and architecture: for a system with 6 accelerators, depending on the cache design, speed-up swings from 0.73× to 6.14×, on average, relative to a baseline sequential system (with a single accelerator and a direct-mapped, 2KB cache with 32B lines). Considering both performance and area, the best architecture is found to be a 4-port multi-pump direct-mapped cache with a 16KB cache size and a 128B line size.**

I. Introduction

Field-programmable gate arrays (FPGAs) have recently been garnering attention for their successful use in computing applications, where they implement custom hardware accelerators specially tailored to a particular application. Indeed, recent work has shown that implementing computations using FPGA hardware has the potential to bring orders of magnitude improvement in energy-efficiency and throughput (e.g. [12]) vs. realizing computations in software running on a conventional processor. In general, such FPGA-based computing systems include a processor, which performs a portion of the work in software, as well as one or more FPGA-based accelerators, which perform the compute- and/or energy-intensive portions of the work. The processor may be a high-performance Intel or AMD `x86` processor running on a connected host PC, or alternately, in the case of embedded systems, a soft processor is often used, implemented on the same FPGA as the accelerators. This paper deals with a key aspect of such embedded processor/parallel-accelerator systems – the memory architecture design.

While a variety of different memory architectures are possible in processor/accelerator systems, a commonly-used approach is one where data *shared* between the processor and accelerators resides in a shared memory hierarchy comprised of a cache and main memory. The advantage of such a model is its simplicity, as cache coherency mechanisms are not required. The disadvantage is the potential for contention when multiple accelerators and/or the processor access memory concurrently. Despite this potential limitation, we use the shared memory model as the basis of our initial investigation, with our results being applicable (in future) to multi-cache scenarios. In our work, data shared among the processor and parallel accelerators is accessed through a shared L1 cache, implemented using on-FPGA memory, and backed by off-chip DDR2 SDRAM. Non-shared local data within a single accelerator is implemented within the accelerator itself using on-FPGA RAMs. We explore the following question: given multiple parallel FPGA-based accelerators that access a shared L1 cache memory, how should the cache and its interface be architected to maximize application performance?

We consider the impact of three traditional cache parameters on performance: 1) cache size; 2) line size; and, 3) associativity. While such parameters have been investigated in standard processors, an FPGA-based cache implementation presents unique challenges. For example, set associativity is typically realized using multiple memory banks with multiplexers to select (based on cache tags) which bank contains the correct data. Multiplexers are costly to implement in FPGAs, potentially making the area/performance trade-offs of associativity different in FPGAs vs. custom chips. Beyond traditional cache parameters, we also consider the hardware design of the FPGA-based cache and its interface.

Dual-ported memory blocks in commercial FPGAs allow a natural implementation of an L1 cache when a single accelerator is present: the processor is given exclusive access to one port, and the accelerator given exclusive access to the second port. With more than two parallel accelerators, the potential for memory contention exists. We explore the extent to which such contention can be mitigated via the cache interface. The accelerators can *share* a memory port with arbitration between concurrent accesses happening outside the memory – in our case, using the Altera Avalon Interface [3]. Alternatively, a multi-ported memory can be used. We consider both approaches in this work, and evaluate two multi-ported cache implementations: 1) multi-pumping, where the underlying cache memory operates at a multiple of the system frequency, allowing multiple memory reads/writes to happen in a single system cycle; and 2) an alternative multi-porting approach recently proposed in [18], comprising the use of multiple RAM banks and a small memory, called the *live-value table*, that tracks which RAM bank holds the most-recently-written value for

978-0-7695-4699-5/12 $26.00 © 2012 IEEE
DOI 10.1109/FCCM.2012.13

a memory address. The two porting schemes offer different performance/area trade-offs.

This paper makes the following contributions:

- Parameterized multi-ported cache memory designs for FPGAs based on two different approaches to multi-porting. Verilog parameters allow easy modification of traditional cache attributes such as cache size, line size and associativity. The designs are open source and freely downloadable (*www.legup.org*).
- The multi-ported caches do not require memory partitioning and allow single-cycle concurrent accesses to all regions of the cache. To the authors' knowledge, this is the first of its kind to be implemented on an FPGA.
- An analysis of the impact of traditional cache parameters on application performance in FPGA-based processor/parallel-accelerator systems.
- An analysis of performance and area trade-offs in multi-ported cache memory design for processor/accelerator systems. To the authors' knowledge, this is the first study of its kind.
- A demonstration of the effectiveness of deploying multiple accelerators, operating in parallel, to improve application performance vs. the single accelerator case.

We conduct our research using the LegUp open source high-level synthesis (HLS) tool [8] from the University of Toronto. We target the Altera DE4 board, containing a Stratix IV 40nm FPGA [13], and show that application performance depends strongly on the cache design and its interface.

The remainder of this paper is organized as follows: Section II presents background and related work. Section III provides an overview of the targeted system. The multi-ported cache designs are described in Section IV. Section V outlines the set of memory architectures explored in this study – 160 in total. An experimental study is described in Section VI. Section VII concludes and offers suggestions for future work.

II. Background

A. Memory Architecture in FPGA-Based Computing

A recent work by Cong et al. proposes memory partitioning as a way of dealing with memory contention imposed by multiple parallel accelerators [11]. The idea is to partition the global memory space based on the regions of memory accessed by each accelerator. Each partition is then implemented as a separate memory bank, with the objective being to minimize the number of accelerators that need concurrent access a particular bank. The partitioning approach yields good results when accelerators operate on separate memory regions – i.e. for applications with memory parallelism. Ang et al. likewise describes a multi-ported cache, where independent sub-caches correspond to separate partitions of the global address space [6]. The techniques described in our paper, however, are independent of the accelerator memory access patterns and thus are orthogonal to the memory partitioning approach. Combining our techniques with memory partitioning is an area for future research. We believe the cache designs evaluated in this paper are also compatible with the recently-proposed CoRAM and LEAP memory architectures [9], [1], both of which provide a memory abstraction for use in FPGA-based computing, and a generic well-defined API/model for managing data transfers between off-FPGA and on-FPGA memory.

Another interesting recent work is the CHiMPS HLS project which uses multiple caches [19]. Each cache corresponds to a particular region of global memory, based on an analysis of a program's access patterns. In CHiMPS, the regions of memory that are "covered" by different caches may overlap, and in such cases, cache coherency is maintained by flushing. One can consider the CHiMPS' memory handling to be that of memory partitioning, permitting duplication. Aside from cache size, [19] does not consider other cache parameters (e.g. line size) as in the current work.

Altera's C2H tool targets a hybrid/processor accelerator system with a cache, however, only the processor has access to the cache (the accelerator can only access off-chip memory), and as such, the cache must be flushed before the accelerator is activated if the two are to share memory [2].

The idea of implementing a parameterizable cache on an FPGA has been proposed in several prior works (e.g. [20], [22]). However, the prior efforts only evaluated single-ported caches in isolation; that is, outside of the parallel accelerator scenario and without a comprehensive multi-benchmark study. Other works (e.g. [15], [10]) have considered implementing configurable caches in the ASIC domain, which presents different trade-offs than FPGAs.

B. LegUp High-Level Synthesis

In this study, the LegUp high-level synthesis tool is used to automatically synthesize `C` benchmark programs to hybrid processor/parallel-accelerator systems [8]. With LegUp, one can specify that certain `C` functions (and their descendants) in a program be synthesized to hardware accelerators, with the balance of the program running in software on a MIPS soft processor [21]. The original versions of the HW-designated functions are replaced by wrapper functions in the software; the wrappers invoke and send/receive data to/from the corresponding accelerators via the Altera Avalon interconnect.

III. System Architecture Overview

Our default system architecture is shown in Fig. 1. It is composed of a MIPS soft processor with one or more hardware accelerators, supported by memory components, including the on-chip dual-port data cache and the off-chip DDR2 SDRAM. Note also that accelerators may have local memory for data not shared with other modules in the system. An instruction cache is instantiated within the MIPS processor as it is only accessed by the processor. The components are connected to each other over the Avalon Interface in a point-to-point manner. This interconnection network is generated by Altera's SOPC Builder tool [4]. The point-to-point connections allow multiple independent transfers to occur simultaneously, which is not feasible with a bus topology. Communication between two components occur via memory-mapped addresses. For example, the MIPS processor communicates with an accelerator by writing to the address associated with the accelerator. When multiple components are connected to a single component,

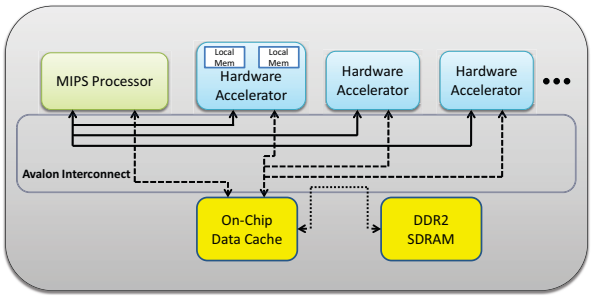

Figure 1. Default system architecture.

such as the on-chip data cache, a round-robin arbiter is automatically created by SOPC Builder.

The solid arrows in Fig. 1 represent the communication links between the processor and accelerators. These links are used by the processor to send arguments to accelerators, invoke accelerators, query an accelerator's "done" status, and retrieve returned data, if applicable. Two modes of execution exist in our system: sequential and parallel. In sequential mode, either the processor or a single accelerator execute at a given time, but not both. Thus, once the processor starts an accelerator, it is stalled until the accelerator finishes. In parallel mode, the processor and multiple accelerators can execute concurrently. The processor first invokes a set of parallel accelerators, then continues to execute without being stalled. The processor may perform other computations, or may check if the accelerators are done by polling on addresses assigned to the accelerators.

The dotted arrows represent communication links between the processor/accelerators and the shared memory hierarchy. The data cache comprises on-chip dual-port block RAMs and memory controllers. On a read, if it is a cache hit, the data is returned to the requester in a *single* cycle. On a miss, the memory controller bursts to fetch a cache line from off-chip DDR2 SDRAM. In our system, this takes 24 cycles for a single burst of 256 bits when there is no contention from other accesses. Depending on the cache line size, the number of bursts is varied. On a burst, after an initial delay of 23 cycles, each additional cycle returns 256 bits of data, continuing until a cache line is filled. As with many L1 caches, we employ a *write-through* cache owing to its simplicity[1].

Note that our approach is *not* that of a single monolithic memory hierarchy. Each accelerator has its own local memory for data that is not shared with the processor or other accelerators. This allows single cycle memory access for all local memories.

IV. Multi-Ported Cache Designs

Current commercial FPGAs contain block RAMs having up to two ports, leading to our default architecture where the processor accesses one port of the RAM and an accelerator accesses the other port. A separate memory controller controls each port. One port is always reserved for the processor, as the processor requires different control signals from the accelerators. Hence, with more than one accelerator, multiple accelerators share the second port as in Fig. 1. This architecture is suitable in two scenarios: 1) sequential execution, where only the processor or a single accelerator is executing at a given time; 2) parallel execution either with a small number of accelerators, or for compute-intensive applications, where accelerators do not access memory often.

For memory-intensive applications, with many accelerators operating in parallel, a dual-ported cache architecture may result in poor performance, as accelerators contend for one port of the cache, leading to the accelerators being stalled most of the time. For increased memory bandwidth, we investigate two types of multi-ported caches, both of which allow multiple concurrent accesses to all regions of the cache in every cycle.

A. Live-Value Table Approach

The first multi-ported cache is based on the work by LaForest et al. [18]. The original work in [18] replicates memory blocks for each read and write port, while keeping read and write as *separate* ports, and uses a live-value table (LVT) to indicate which of the replicated memories holds the most recent value for a given memory address. Each write port has its own memory bank containing R memories, where R is the number of read ports. On a write, the writing port writes to *all* memories in its bank, and also writes its port number to the corresponding memory address in the LVT, indicating that it is the most-recent writer to the address. Read ports connect to one memory block in each write-port bank. On a read, the reading port reads from all of its connected memories, and looks up the memory address in the LVT, which returns the previously-written port number. This is used to select the most-recently-written value from one of the connected memories. The LVT is implemented with registers, as multiple ports can read and write from different memory locations at the same time. The original work was intended for register files in processors with *separate* read and write ports, and uses a *simple* dual-port memory, where one port is reserved for writes and the other port is reserved for reads. With this architecture, the total memory consumed is $\alpha \times$ the original memory size, where α is equal to the number of write ports $\times$ the number of read ports.

In the case of caches, the number of read and write ports is equal, and with n read/write ports, the cache size would grow by n^2. However, in our system, the read and write ports do not need to be separate. A read and a write port can be combined into a single read/write port, since an accelerator can only read or write at a given time, but not do both. One *true* dual-ported memory can therefore be used for two read/write ports, instead of using 2 simple dual-ported memories with 2 read ports and 2 write ports. This reduces the total memory consumption to less than half of that in [18]. A 4-ported cache using the new architecture is shown in Fig. 2, where each *M* represents a true dual-ported memory, and *MC* represents a memory controller. For clarity, only the output (read) lines are shown from each memory block. The input (write) lines are connected from

[1]Write-through caches do not require bookkeeping to track which cache lines are dirty.

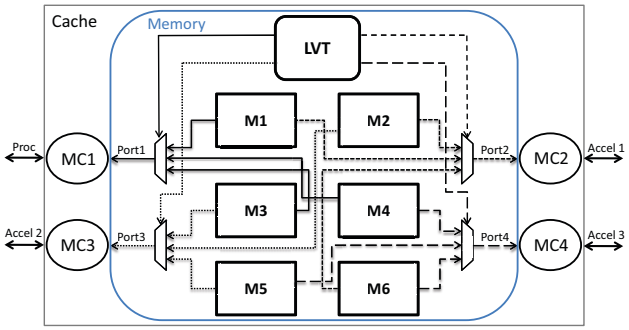

Figure 2. LVT-based 4-ported cache.

each port to the same memory blocks as shown by the arrows (without the multiplexer). A memory controller is connected to each port of the memory, which is subsequently connected to either the processor or an accelerator.

In our variant of the LVT memory approach, it is required that any two ports have one memory block in common. For example, in Fig. 2, port 1 shares memory blocks M1, M3 and M4, with ports 2, 3, and 4, respectively. This allows data written by port 1 to be read by all other ports. On a write, a port writes to all of the memory blocks that it is connected to, which is *n-1* blocks. As in the original LVT memory implementation, the port also writes to the LVT to indicate it has updated a memory location most recently. On a read, a port reads from all connected RAM blocks and selects the data according to the port number read from the LVT. Compared to the previous work which caused memory to grow by $n^2\times$ with n ports, our LVT variant scales as:

$$\textit{New cache size} = \frac{n \times (n - 1)}{2} \times \textit{original cache size} \quad (1)$$

The 4-port cache in Fig. 2 replicates memory size by $6\times$, whereas the approach in [18] replicates memory size by $16\times$. The output multiplexer, which selects between the memory blocks is also reduced from an *n-to-1* multiplexer to an *(n-1)-to-1* multiplexer.

This new multi-ported cache based on the LVT approach is referred to as the *LVT cache* in this paper. This can be compared to multi-cache architectures, where multiple caches are distributed, with each cache connected to a processor or an accelerator, and a cache coherency scheme is implemented to keep the memory synchronized. The main advantage of the LVT cache architecture is that it offers a shared memory space, which acts as one combined piece of memory, thus no cache coherency scheme is needed, avoiding area and latency costs for synchronization. This multi-ported cache allows all ports to access coherent data concurrently every cycle.

Note that since the cache line sizes can be large, the LVT cannot keep track of every byte in the cache. Hence the LVT works on a cache line granularity, to keep track of which cache line was the most recently updated. This can cause *cache line write conflicts*, when two ports attempt to write to the same cache line in the same cycle, even if they are writing to different bytes in the line. To resolve this,

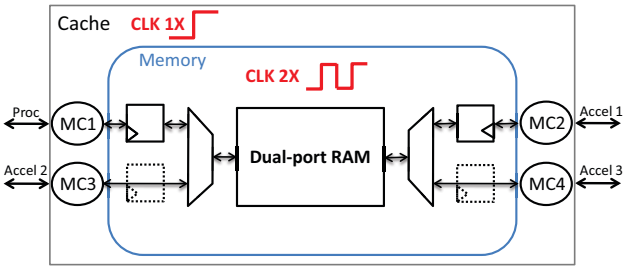

Figure 3. 4-ported cache with double-pumping.

an arbiter is implemented, which serializes the writes to the same cache line from multiple ports. This is only for cache writes to the same cache line. Any reads from the same line or writes to independent cache lines can occur concurrently.

B. Multi-Pumping

Modern FPGAs, such as the Altera Stratix IV, have memory blocks which can operate at over 500 MHz [5] – a speed which is often much faster than the speed of the overall system. We use this advantage to create another type of multi-ported cache, using dual-ported memories clocked at twice the system speed – a well-known technique called memory *multi-pumping*. In one clock cycle of the system clock, the double-pumped dual-ported memories can perform 4 memory accesses, which from the system's perspective, is equivalent to having a 4-ported memory. This architecture is shown in Fig. 3. Four concurrent accesses are broken down into two sets of two accesses. The first set of accesses is presented to the RAM immediately, while the second set is stored in registers. In the second-half of a system cycle, the results of the first set of accesses are stored in registers, while the second set of accesses is presented to the RAM. At the end of a complete system cycle, both sets of accesses are complete. In essence, by running the RAM at $2\times$ the system clock frequency, multi-pumping mimics the presence of a RAM with $2\times$ as many ports as there are actually present in the hardware.

The advantage of this approach is that it does not increase memory consumption. It uses the same amount of memory as a single dual-ported cache with extra control logic and registers to steer the data in and out of the RAMs. The limitation, however, is that the system needs to run slow enough so that the memory can run at multiples of the system clock. This constraint usually makes it difficult to run the memories at more than twice the system clock. In addition to its reduced memory consumption, the multi-pumping cache bears other advantages over the LVT approach. Since memory blocks are not replicated, no extra logic is required to keep track of which memory block holds the most recent value, thus eliminating the LVT table. This in turn allows us to make use of byte enable signals to only write to the bytes of interest, instead of writing the entire cache line. Thus, this approach does not have the *cache line write conflicts*, described previously. Even if two ports are writing to the same cache line, no arbitration is needed unless they are writing to the same bytes.

The multi-pumping cache uses less logic and memory

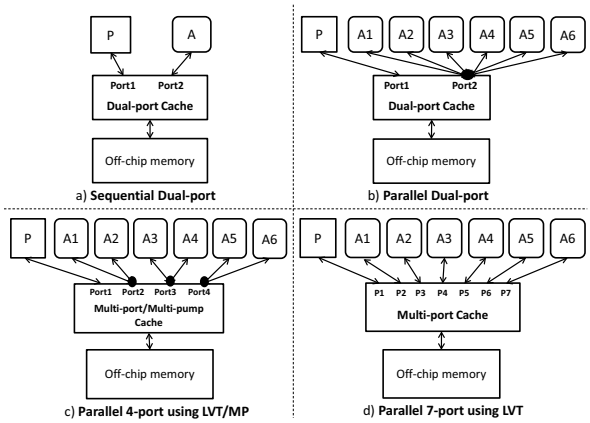

Figure 4. Architectures evaluated in this work.

resources than the LVT cache, which often leads to higher Fmax (shown in Section VI), with the caveat that multi-pumping cannot be scaled to more than 4-ports in our system. The multi-ported cache based on the multi-pumping approach is referred to as the *MP cache* for the rest of this paper.

V. System Architectures Explored

Using the LVT and MP caches, we investigated 5 different systems, shown in Fig. 4, where *P* denotes the processor, and *A* denotes an accelerator. Fig. 4(a) shows our default architecture, where a single accelerator executes sequentially to perform all of the work in the program while the processor is stalled. Fig. 4(b) illustrates the same cache architecture as Fig. 4(a), but with multiple accelerators which execute in parallel. With one of the ports designated for the processor, all of the accelerators arbitrate for the second port (via Avalon). Fig. 4(c) shows a 4-ported cache architecture, using either an LVT or an MP cache, with two accelerators sharing one port of the cache – Avalon arbitrates between the two accelerators that share a port. Fig. 4(d) shows an architecture where the processor and each accelerator has its own port to the cache. This has the highest possible memory bandwidth, since no contention exists to the cache and all of the cores can access memory at the same time. Contention still exists to access off-chip memory. Only the LVT cache can be used for the architecture in Fig. 4(d), as the MP memory with its surrounding control logic cannot run at more than $2\times$ the system clock. For the parallel cases, 6 accelerators were chosen to evaluate performance variations with different cache architectures, ranging from high memory contention to no memory contention to access the cache.

For each of the 5 different systems, we explore a variety of cache configurations, shown in Table I. Four different cache sizes are investigated, and for each cache size, 4 different line sizes are studied. For each cache and line size, we lastly investigate direct-mapped (one-way) caches vs. 2-way set-associative caches. Hence, we evaluate a total of $5 \times 4 \times 4 \times 2 = 160$ different memory architectures across 9 parallel benchmarks (c.f. Section VI).

Table I
Cache configurations evaluated.

Cache Size	**Cache line size / # of Cache lines**			
2KB	32B/64	64B/32	128B/16	256B/8
4KB	32B/128	64B/64	128B/32	256B/16
8KB	32B/256	64B/128	128B/64	256B/32
16KB	32B/512	64B/256	128B/128	256B/64
Associativity	Direct-mapped		2-way Set-associativity	

VI. Experimental Study

A. Benchmarks

The 9 parallel benchmarks used in this study are all memory-intensive *data-parallel* programs, meaning that no two accelerators write to the same memory address, but may read from the same address. Each benchmark includes predefined inputs and golden outputs, with the computed result checked against each golden output at the end of the program.

- The array addition benchmark sums the total of 6 arrays, each with 10,000-integer elements.
- The box filter is a commonly-used kernel in image processing. It has the effect of smoothing an image, by using a 3×3 filter to eliminate any out-of-place pixel values.
- The dot product benchmark does pairwise multiplication of each element in two 6,000-element integer arrays and calculates the total sum.
- The GSM$\times$6 benchmark is adopted from the CHStone benchmark [16] and performs 6 linear predictive coding analyses of a global system for mobile communications.
- The histogram benchmark takes an input of 36,000 integers between 1 to 100, and accumulates them into 5 equally-sized bins.
- The line of sight benchmark uses the Bresenhams line algorithm [7] to determine whether each pixel in a 2-dimensional grid is visible from the source.
- The matrix multiply benchmark multiplies a 100×6 matrix with a 6×100 matrix.
- The matrix transpose benchmark transposes a 1024×6 matrix.
- The perfect hash benchmark hashes 24,000 integers between 0 and 500,000,000 and creates a perfect hash table.

Note that for all of benchmarks the input data is sized to be equally divisible by 6, as 6 parallel accelerators are used, with each accelerator performing an equal amount of work. This can easily be changed to handle different numbers of accelerators.

Each benchmark was first simulated on each type of system architecture for each *cache-size/line-size/associativity* using a ModelSim functional simulation. The total number of execution cycles was extracted from the ModelSim simulation, using an accurate simulation model for the off-chip DDR2 SDRAM on the Altera DE4 board. Following cycle-accurate simulation, each benchmark was synthesized to Altera Stratix IV with Quartus II (ver. 10.1SP1) to obtain area and critical path delay (Fmax) numbers. For Fmax, we use slow 900mV 85 deg. C timing models. Execution

Table II
BASELINE SYSTEM RESULTS.

Time	Cycles	Fmax	Area	Memory
2,028.3 μs	259,216	127.8 MHz	13,460 ALMs	301,693 bits

time for each benchmark is computed as the product of execution cycles and post-routed clock period. Exploring all benchmarks and architectures required a total of 1,440 ModelSim simulation *and* Quartus synthesis runs, which was done using SciNet, a top-500 supercomputer [14].

B. Results

Figs. 5, 6 and 7 show average execution time (speed-up), cycle count (speed-up) and Fmax, respectively, relative to a baseline system which employs a single accelerator running sequentially using a 1-way (direct-mapped) 2KB cache with a 32-byte line size. The geometric mean results for the baseline system (across all benchmarks) are shown in Table II.

Beginning with Fig. 5, observe that larger cache sizes generally provide higher speed-ups, as expected. We see a clear trend in the figure: two clusters of speed-up curves – one cluster clearly above the other. The cluster of six curves with the higher speed-ups corresponds to the three multi-port cache designs considered (each direct-mapped or 2-way set associative). The best speed-ups of 6.14× and 6.0× occur for the parallel 7-port LVT 1-way and the parallel 4-port MP 1-way, respectively. From the bottom cluster of curves, it is evident that, even with 6 accelerators, using a dual-port cache and relying on the Avalon interconnect for arbitration provides poor speed-up results (even slower than sequential execution in some cases!).

Observe in Fig. 5 that performance varies widely as line sizes are increased. In general, performance increases up to 64-byte line size for cache sizes of 2 and 4 KBs, and 128-byte line size for cache sizes of 8 and 16 KBs. For a given cache size, as line size increases, the total number of lines are reduced. With a smaller number of cache lines, a greater number of memory addresses map to the same cache line, making the line more likely to be evicted for a new cache line. Miss rate actually goes up if the line size is too large relative to the cache size [17]. With multiple accelerators accessing memory, this can cause *cache thrashing* to occur, where the same set of cache lines are evicted and retrieved continuously, causing excessive off-chip memory accesses. Larger lines sizes also lead to longer fetch cycles. Indeed, smaller line sizes perform better for smaller caches as shown in Fig. 6. This effect is mitigated with 2-way set-associative caches, and also as cache sizes themselves become bigger. In Fig. 6, 4-port MP is overlapped with 4-port LVT for both 1-way and 2-way, with 4-MP showing slightly better results due to *cache write conflicts*, discussed previously. Sequential systems show less sensitivity to cache size and line size, as only one accelerator is accessing memory sequentially.

Fig. 7 shows the impact of multiplexers on Fmax. The most drastic decrease in Fmax occurs for 2-way systems, as they require the most multiplexing logic. A 2-way set-associative cache instantiates two different sets of memories, with multiplexers on the inputs and outputs to select between the two sets. In the case of multi-ported memories, extra multiplexers and/or replicated memory blocks exist within each set. As expected, the 7-port LVT cache shows the greatest decrease in Fmax as it uses the highest number of replicated memories with the largest output MUX. This is followed by the 4-port LVT and MP caches. On average, parallel dual-port systems show lower Fmax than sequential and 4-ported systems since 6 accelerators connect to 1 port of the cache, creating a larger MUX (6-to-1) than other multi-ported systems. Accelerators generated by LegUp HLS for some benchmarks, such as matrix multiply, exhibited long combinational paths to the cache, which in combination with the 6-to-1 MUX, cause a significant impact on Fmax. In general, Fmax is also decreases with increasing line sizes as larger MUXes are required to select specific data from longer lines.

Turning to area (# of ALMs) in Fig. 8, all of the 2-way systems consume more area than their respective 1-way systems, as expected[2]. Among all systems, the 7-ported architecture has the largest size, with the most MUXes, followed by the 4-ported LVT architecture. This has more area than the 4-ported MP architecture due to its LVT table, as well as output MUXes to select between its replicated memory blocks. All of the parallel systems consume more area than sequential systems due to their having multiple accelerators. Area also increases with increasing line size due to bigger MUXes.

In terms of memory consumption, shown in Fig. 9, the 7-port LVT 2-way consumes the most memory bits, owing to its replicated memories. Note that the graph illustrates the total memory consumption of the system, which include the data cache, instruction cache, local accelerator memories, as well as any memory consumed by other components such as the DDR2 controller, and dividers. The reason that the memory consumption decreases slightly with increasing line sizes is due to the tags and the valid bit, which is stored as part of the each line in addition to the data. Hence for a given cache size, the total number of lines are reduced with bigger line sizes, reducing tag/valid-bit overhead.

Note that 7-port LVT 2-way, 7-port LVT 1-way, and 4-port LVT 2-way have missing data points in the figures, as they cannot fit within the Stratix IV on the DE4 board for larger line sizes. This is due to the structure of the Stratix IV M9K memory blocks, that can store 9,216 bits. Each M9K block in true dual-port mode can operate with 18-bit-wide words. A total of 1,280 M9K blocks are available on the Stratix IV [5]. Thus, a 256 byte line size with tags/valid bit consumes 115 M9K blocks. For a 7-ported LVT 1-way cache, where the memory blocks are replicated by 21× (see Eqn. 1), this totals 2,415 M9K blocks, exceeding the number available. Despite the fact that a 7-port LVT 1-way cache using a 256B line size only consumes a small fraction of the total available memory bits on the device, it cannot fit due to the architecture of the dual-ported M9K blocks. Similar conditions arise for 7-port LVT 2-way, and 4-port LVT 2-way architectures with large line sizes.

[2]The area results represent the complete system including the processor and accelerators.

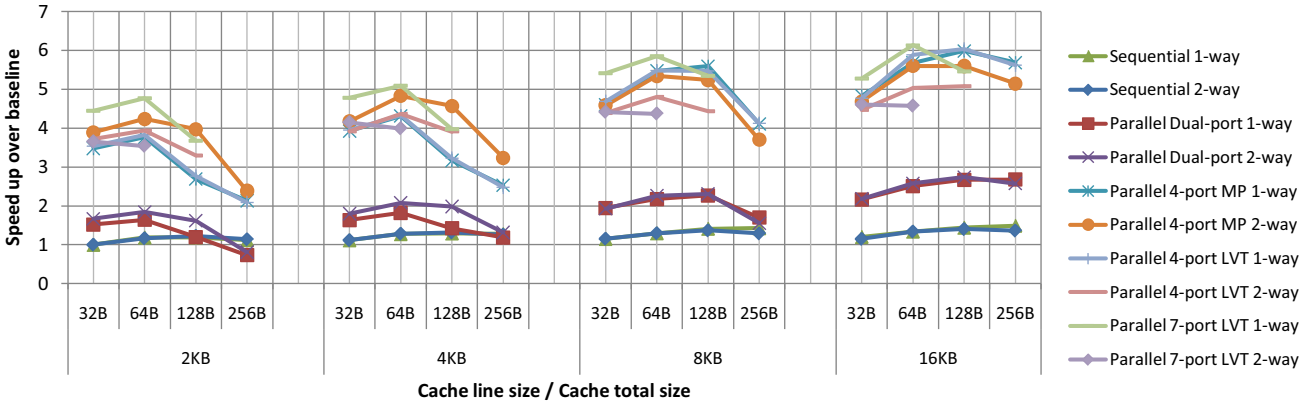

Figure 5. Execution time (geometric mean).

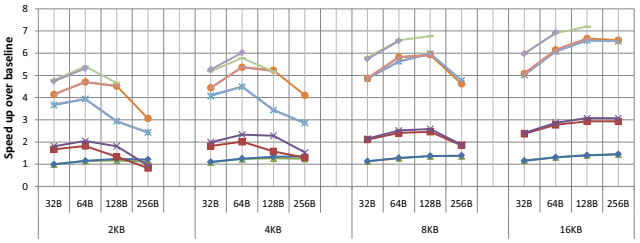

Figure 6. Execution cycles (geometric mean).

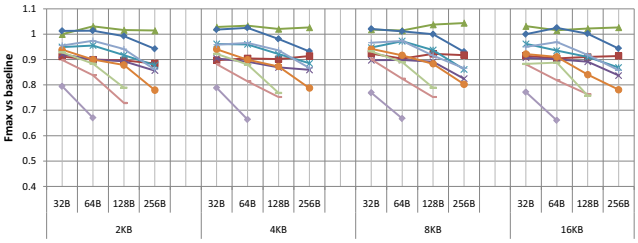

Figure 7. Fmax (geometric mean).

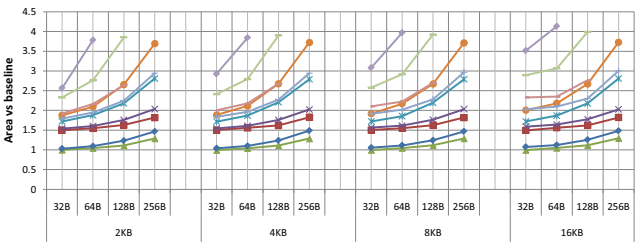

Figure 8. Area in Stratix IV ALMs (geometric mean).

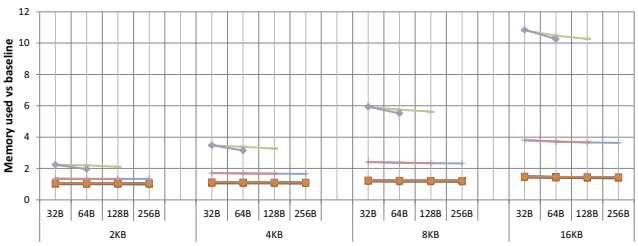

Figure 9. Memory consumption (geometric mean).

Table III shows the performance of each benchmark in 4 cases: 1) The *worst* result for the benchmark listed in the top line; 2) the *best* result for the benchmark shown in the second line; 3) the result for the benchmark using the cache architecture which yields the *best* performance in geomean results over all benchmarks: the 7-port LVT 1-way cache with 16KB cache and 64-byte line size, shown in the third line; and lastly, 4) the result with the cache architecture which yields the *second best* performance in geomean results: the 4-port MP 1-way cache with 16KB cache and 128-byte line size. The data in Table III only includes parallel 6-accelerator architectures; i.e. the gains shown are isolated to be due to memory changes and are not due to parallelization. The 'Cache Arch.' column is organized as *type of portedness/cache size/line size/associativity*. For example, 2-port/2KB/256B/1 indicates a Parallel Dual-port 1-way cache architecture with a 2KB cache and a 256-byte line size. The 'Time' column shows the execution time in μs, with the number in the brackets indicating the speed-up compared to the worst-case.

The dot product benchmark shows the biggest improvement, with 29.4$\times$ speed-up between its best and worst cases. This is due to the cache line thrashing effect, described previously. The GSM$\times$6 benchmark shows the least sensitivity to the cache design, with the gap between the best and worst being 2.6$\times$. Further analysis showed the GSM benchmark to be the least memory intensive among all benchmarks. It is interesting to note, however, that the biggest cache size does not always yield the best results, as shown by the box filter, dot product, GSM$\times$6, and the histogram benchmarks. Moreover, observe that no single cache architecture is best for all benchmarks.

In summary, the *7-LVT/16KB/64B/1* architecture shows comparable results to each best case on a per-benchmark basis. However, with much less area and memory consumption, the *4-MP/16KB/128B/1* architecture also exhibits

Table III
INDIVIDUAL BENCHMARK RESULTS.

Benchmark	Cache Arch.	Time	Cycles	Fmax
Add	2-Port/2KB/256B/1	10,596 (1)	1,438,565	135.8
	4-MP/16KB/256B/1	443 (23.9)	61,229	138.1
	7-LVT/16KB/64B/1	524 (20.2)	66,868	127.6
	4-MP/16KB/128B/1	507 (20.9)	67,682	133.58
Box Filter	2-Port/2KB/256B/1	11,635 (1)	1,082,204	93.0
	4-MP/4KB/64B/2	947 (12.3)	114,785	121.2
	7-LVT/16KB/64B/1	1,066 (10.9)	118,446	111.1
	4-MP/16KB/128B/1	1,181 (9.86)	128,775	109.08
Dot Product	2-Port/2KB/256B/1	2,901 (1)	390,864	134.7
	7-LVT/8KB/128B/1	99 (29.4)	11,167	113.2
	7-LVT/16KB/64B/1	100 (29.1)	12,837	128.6
	4-MP/16KB/128B/1	104 (27.8)	13,851	132.73
GSM×6	2-Port/4KB/32B/2	246 (1)	21,903	89.0
	4-MP/4KB/32B/1	94 (2.6)	10,493	111.5
	7-LVT/16KB/64B/1	108 (2.3)	9,626	89.1
	4-MP/16KB/128B/1	105 (2.4)	10,119	96.61
Histogram	2-Port/4KB/32B/2	1,465 (1)	187,882	128.3
	4-LVT/2KB/128B/1	476 (3.1)	64,545	135.6
	7-LVT/16KB/64B/1	560 (2.6)	63,459	113.3
	4-MP/16KB/128B/1	539 (2.7)	64,545	119.76
Line of Sight	2-Port/2KB/256B/1	6,139 (1)	597,016	97.3
	7-LVT/16KB/64B/1	376 (16.3)	43,658	116.0
	7-LVT/16KB/64B/1	376 (16.3)	43,658	116.0
	4-MP/16KB/128B/1	394 (15.6)	44,934	114.2
Matrix Mult	2-Port/2KB/256B/1	836 (1)	68,803	82.3
	7-LVT/16KB/128B/1	55 (15.2)	5,521	100.4
	7-LVT/16KB/64B/1	58 (14.3)	5,983	102.7
	4-MP/16KB/128B/1	62 (13.4)	6,161	98.6
Matrix Trans	2-Port/4KB/256B/2	1,790 (1)	217,357	121.4
	4-LVT/16KB/128B/1	178 (10.1)	22,623	127.4
	7-LVT/16KB/64B/1	228 (7.8)	25,505	111.7
	4-MP/16KB/128B/1	188 (9.5)	22,623	120.5
Perfect Hash	2-Port/2KB/256B/1	19,732 (1)	2,740,215	138.9
	4-MP/16KB/64B/2	2,451 (8.1)	332,281	135.6
	7-LVT/16KB/64B/1	2,799 (7.1)	357,306	127.7
	4-MP/16KB/128B/1	3,621 (5.5)	464,224	128.2

competitive results to the *7-LVT/16KB/64B/1* architecture, sometimes performing better depending on the benchmark. Recall that multi-pumping does not require the cache to be replicated and does not use the LVT which requires bigger MUXes, making the total area approximately half, with a cache size 21× smaller than the 7-LVT (see Figs. 8 and 9). Thus, if the designer were forced into choosing a memory architecture without knowledge of an application's specific access patterns, the *4-MP/16KB/128B/1* architecture appears to be a good choice.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we explored a wide range of cache architectures in the context of FPGA-based processor/parallel-accelerator platform. We introduced two types of shared multi-ported caches which do not require memory partitioning and allow single-cycle concurrent accesses to all regions of the cache. The *multi-pumping cache* uses less logic and does not increase memory consumption, which is suitable for memory/area-constrained designs. With enough resources, the *LVT cache* can allow a larger number of ports for maximum memory bandwidth. While our results with traditional cache parameters conform to other research in the computer architecture domain, an FPGA implementation poses challenges which are unique to the platform. Specifically, multiplexers, which are required to implement different cache line sizes, associativity, and portedness are expensive in FPGAs and significantly impact performance results. On the other hand, high-speed on-chip block RAMs which can often run much faster than their surrounding system offer a differentiating advantage compared to other computing platforms – they make multi-pumping a viable approach in FPGAs. We conclude that the best architecture from a performance angle, considering all aspects of total execution time, area, and memory consumption, is the 4-port multi-pump direct-mapped cache with a 16KB cache size and a 128 byte line size. We believe this architecture to be the best choice for systems which run under 200 MHz.

Future work includes combining the memory architectures analyzed here with the work on memory partitioning in HLS (e.g. as described in [11]). We are also interested in analyzing these architectures from an energy-efficiency perspective.

REFERENCES

[1] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP scratchpads: automatic memory and cache management for reconfigurable logic. In *ACM Int'l Symp. on FPGAs*, pages 25–28, 2011.

[2] Altera, Corp., San Jose, CA. *Nios II C2H Compiler User Guide*, 2009.

[3] Altera, Corp., San Jose, CA. *Avalon Interface Specification*, 2010.

[4] Altera, Corp., San Jose, CA. *SOPC Builder User Guide*, 2010.

[5] Altera, Corp., San Jose, CA. *TriMatrix Embedded Memory Blocks in Stratix IV Devices*, 2011.

[6] S.-S. Ang, G. Constantinides, P. Cheung, and W. Luk. A flexible multi-port caching scheme for reconfigurable platforms. In *Applied Reconfigurable Computing*, pages 205–216, 2006.

[7] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 1965.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *ACM Int'l Symp. on FPGAs*, pages 33–36, 2011.

[9] E. Chung, J. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *ACM Int'l Symnp. on FPGAs*, pages 97–106, 2011.

[10] J. Cong, K. Gururaj, Hui Huang, Chunyue Liu, G. Reinman, and Yi Zou. An energy-efficient adaptive hybrid cache. In *IEEE Int'l Symp. on Low-Power Electronics and Design*, pages 67–72, 2011.

[11] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 16, April 2011.

[12] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):1–29, 2009.

[13] DE4, Altera Corp, San Jose, CA. *DE4 Development and Education Board*, 2012.

[14] C. Loken et al. SciNet: Lessons learned from building a power-efficient top-20 system and data centre. *Journal of Physics: Conference Series*, 256, 2010.

[15] A. Gordon-Ross, F. Vahid, and N.D. Dutt. Fast configurable-cache tuning with a unified second-level cache. *IEEE Trans. on VLSI*, pages 80–91, 2009.

[16] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[18] C.E. LaForest and J.G. Steffan. Efficient multi-ported memories for FPGAs. In *ACM Int'l Symp. on FPGAs*, pages 41–50, 2010.

[19] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig. Performance and power of cache-based reconfigurable computing. In *Int'l Symp. on Computer Architecture*, pages 395–405, 2009.

[20] A. Santana Gil, J. Benavides Benitez, M. Hernandez Calvino, and E Herruzo Gomez. Reconfigurable cache implemented on an FPGA. In *IEEE Int'l Conf. on Reconfigurable Computing*, pages 250–255, 2010.

[21] University of Cambridge. *The Tiger "MIPS" processor (http://www.cl.cam.ac.uk/teaching/ 0910/ECAD+Arch/mips.html).*, 2010.

[22] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *IEEE Int'l Conf. on Field-Programmable Technology*, pages 324–327, 2003.