2019 29th International Conference on Field Programmable Logic and Applications (FPL)

Analysis and Optimization of I/O Cache Coherency Strategies for SoC-FPGA Device

Seung Won Min
Electrical and Computer Engineering
University of Illinois
Urbana, USA
min16@illinois.edu

Sitao Huang
Electrical and Computer Engineering
University of Illinois
Urbana, USA
shuang91@illinois.edu

Mohamed El-Hadedy
Electrical and Computer Engineering
California State Polytechnic University
Ponoma, USA
mealy@cpp.edu

Jinjun Xiong
IBM T.J. Watson Research Center
Yorktown Heights, USA
jinjun@us.ibm.com

Deming Chen
Electrical and Computer Engineering
University of Illinois
Urbana, USA
dchen@illinois.edu

Wen-mei Hwu
Electrical and Computer Engineering
University of Illinois
Urbana, USA
w-hwu@illinois.edu

***Abstract*—Unlike traditional PCIe-based FPGA accelerators, heterogeneous SoC-FPGA devices provide tighter integrations between software running on CPUs and hardware accelerators. Modern heterogeneous SoC-FPGA platforms support multiple I/O cache coherence options between CPUs and FPGAs, but these options can have inadvertent effects on the achieved bandwidths depending on applications and data access patterns. To provide the most efficient communications between CPUs and accelerators, understanding the data transaction behaviors and selecting the right I/O cache coherence method is essential. In this paper, we use Xilinx Zynq UltraScale+ as the SoC platform to show how certain I/O cache coherence method can perform better or worse in different situations, ultimately affecting the overall accelerator performances as well. Based on our analysis, we further explore possible software and hardware modifications to improve the I/O performances with different I/O cache coherence options. With our proposed modifications, the overall performance of SoC design can be averagely improved by 20%.**

***Index Terms*—FPGA, heterogeneous computing, cache, cache coherence**

I. Introduction

Heterogeneous SoC-FPGA platforms such as Xilinx Zynq UltraScale+ MPSoC provide flexible development environment with tightly-coupled interfaces between different processing units inside. Depending on the needs of users, these processing units can be combined and programmed to provide the most suitable configuration. For the different components to operate seamlessly together, it is important to understand how data coherency between them are managed. For the traditional server or desktop class machines, there is little meaning of configuring the host system's I/O cache coherence for general FPGA designers because often: 1) manufacturers do not provide any documentations of that level of detail or 2) I/O cache coherence is enabled by default in such scales of systems. On the other hand, in SoC-FPGA design, all available I/O cache coherence options are fully disclosed to the FPGA designers and the designers are responsible of choosing the most suitable methods for target applications.

However, choosing the right I/O cache coherence method for different applications is a challenging task because of it's versatility. By choosing different methods, they can introduce different types of overheads. Depending on data access patterns, those overheads can be amplified or diminished. This versatility not only makes designers hard to decide which methods to use, but also can mislead them to wrong decisions if performance evaluations are incomprehensive. In this work, we analyze the effects of using different I/O cache coherence methods in SoC-FPGA as detail as possible and provide general guide of using each method. Our I/O cache coherence performance analysis consists of software costs and hardware costs and these costs are combined to evaluate the total cost of each method. The contributions of this paper can be summarized as follows:

- Evaluate software and hardware costs of using different I/O cache coherence methods.
- Introduce several optimization techniques which can eliminate some I/O cache coherence costs.
- Provide a complete guide of achieving efficient I/O cache coherence based on real hardware evaluation results.

The rest of the paper is organized as follows. In Section II, we explain backgrounds of different I/O cache coherence strategies. In Section III, we elaborate our experiment environment. In Section IV, we show our software and hardware I/O cache coherence cost evaluation results. In Section V, we provide a general guide of I/O cache coherence optimizations. Section VI discusses related works. Finally in Section VII, we summarize our work and conclude this paper.

II. I/O Cache Coherence

In a modern system design, it is common to use memory as a shared buffer to transfer data between CPUs and I/O devices [1]. However, with CPU caches, data coherency issues

Extended version of this article can be found at arxiv.org/abs/1908.01261

1946-1488/19/$31.00 ©2019 IEEE
DOI 10.1109/FPL.2019.00055

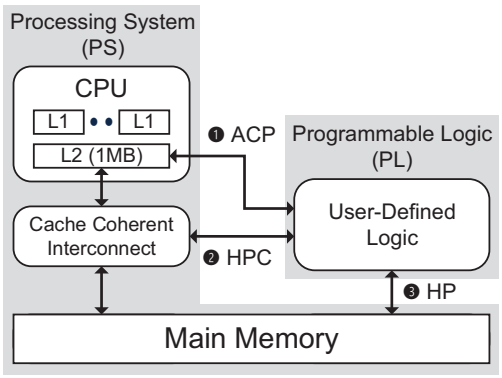

Fig. 1. Simplified block diagram of possible I/O configurations in Xilinx Zynq UltraScale+.

can occur between devices and CPUs. To avoid the situation, different I/O cache coherence methods can be used.

1) Allocating Non-cacheable Memory: The simplest way of achieving I/O cache coherency is making memory accesses non-cacheable. This does not need to be enforced globally, and it can be narrowed down to specific memory regions which are shared between CPUs and I/O devices by setting appropriate ISA-dependent virtual page attributes.

2) Software I/O Coherency: Software I/O coherency requires CPUs to manually flush or invalidate cache lines by executing cache maintenance instructions before any data transactions between CPUs and I/O devices are made. In this method, CPUs can still cache data from the memory regions shared with I/O devices.

3) Hardware I/O Coherency: Hardware coherency relies on hardware implementations included in host systems which let I/O devices to snoop CPU caches. Achieving the cache snooping can be largely done in two ways. First, I/O buses between CPUs and I/O devices can be modified so every memory access requests from I/O devices cause cache snoop requests as well. The second way is directly connecting I/O devices to caches. In this case, I/O devices generate cache snooping requests like other CPU cores.

III. Experiment Environment

All experiments in this paper are done based on Xilinx Zynq UltraScale+ MPSoC. Zynq Ultrascale+ has Processing System (PS) block and Programmable Logic (PL) block as described in Fig. 1. Between the two blocks, there are several types of I/O available. ❶ Accelerator Coherency Port (ACP) interface can access shared L2 cache (1MB) directly. ❷ High Performance Coherent (HPC) interface goes through coherent I/O bus where it can issue cache snooping requests to the CPU cache. ❸ High Performance (HP) interface goes to memory directly and I/O cache coherence should be dealt by the CPU. All interfaces are 128-bit wide and we fix interface frequencies to 300 MHz throughout our experiments, providing the maximum theoretical bandwidths of 4.8 GB/s. Table I summarizes overall Zynq UltraScale+ interfaces and possible I/O cache coherence methods. In the rest of the paper, we refer the HP interface with non-cacheable and cacheable memory allocations as HP (NC) and HP (C), respectively.

Software I/O coherency implementation is embedded in Xilinx drivers and the drivers are capable of identifying the buffer allocation types. If the buffers are allocated as non-cacheable, the drivers do not manually flush or invalidate caches. If the buffers are allocated as cacheable, the drivers automatically perform cache flushes and invalidations.

TABLE I
Available PL Interfaces and Data Coherency Methods in Zynq UltraScale+

Alias	Inteface	Memory Allocation	Data channel is connected to	Coherency Method
HP (NC)	HP	Non-cacheable	Memory	Not Required
HP (C)	HP	Cacheable	Memory	Cache Inst.
HPC	HPC	Cacheable	Memory & Cache (Read-only)	H/W Coherent
ACP	ACP	Cacheable	Cache	H/W Coherent

IV. I/O Cache Coherence and SoC-FPGA

In this section, we evaluate hardware and software costs of different I/O cache coherence methods. For the hardware cost, we are interested in identifying how much the extra steps required to resolve cache snoop requests in hardware can negatively affect I/O bandwidths. For the software cost evaluation, we measure CPU overheads added when hardware coherent I/O interfaces are not supported.

A. Hardware Cost Evaluation

In this experiment, we measure raw bandwidths of non-hardware coherent I/O (HP) and hardware coherent I/O (HPC and ACP) interfaces. The raw bandwidth here means the pure interface bandwidths without any software overheads included. To measure CPU to PL (TX) and PL to CPU (RX) bandwidths, we program PL to initiate data transfers and count how many bus clock cycles spent. For the hardware coherent I/Os, we'd like to also know if there are any bandwidth differences when the shared buffer data for both TX and RX cases are cached or not. To achieve this, we intentionally read/write or flush the entire range of the shared buffers before the data transfers begin. The summary of the test setups can be found at Table II. We do not differentiate between HP (NC) and HP (C) in this experiment as their differences are only at software costs.

TABLE II
Raw Bandwidth Test Setup

Direction	Interface	Before data transfer the buffer has been
CPU ↓ PL	HP	
	HPC (w/ Write)	Written
	HPC (w/ Flush)	Flushed
	ACP (w/ Write)	Written
	ACP (w/ Flush)	Flushed
PL ↓ CPU	HP	
	HPC (w/ Read)	Read
	HPC (w/ Flush)	Flushed
	ACP (w/ Read)	Read
	ACP (w/ Flush)	Flushed

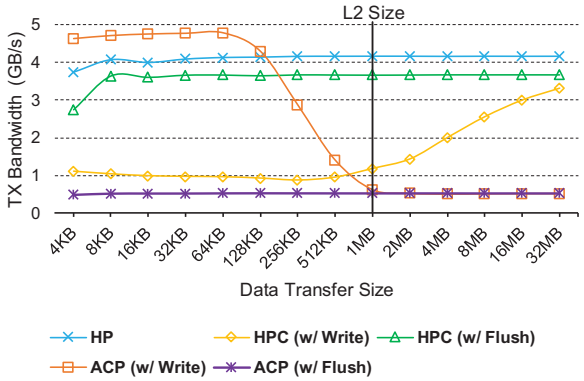

Fig. 2. I/O bus TX (CPU→PL) bandwidth comparison. No software overhead is included in this measurement.

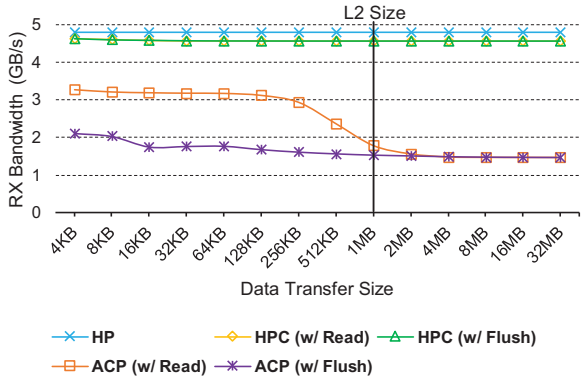

Fig. 3. I/O bus RX (PL→CPU) bandwidth comparison. No software overhead is included in this measurement.

Fig. 2 shows the TX bandwidth measurement results. Starting from the HP results, we observe almost no differences in TX bandwidths while sweeping from 4KB to 32MB data transfers. There is a small bandwidth drop at 4KB due to the initial DRAM access latency, but the overhead of the latency becomes almost not visible as the data transfer size increases.

In case of HPC, we see huge differences when the data is cached or not. For HPC (w/ Flush), there is only a small bandwidth drop compared to HP, but for HPC (w/ Write), the TX bandwidth decreases significantly. Writing larger amount of data to the buffer attenuates this problem as the maximum amount of cached data is limited by the L2 size.

ACP bandwidth is nearly reaching 4.8 GB/s with small sizes of data, but it starts to sharply drop as the data size approaches toward the L2 size. A53 L2 cache does not have hardware prefetching unit and therefore all cache accesses without pre-populated cache lines need to pay cache miss penalties. When the buffer is completely flushed before the data transfer, ACP constantly suffers from the low bandwidth as all cache accesses cause cache misses.

For the RX bandwidth measurement results, we do not see any significant bandwidth changes beside ACP. In Fig. 3, both HP and HPC are reaching near 4.8 GB/s of bandwidths in all cases. In case of ACP, we observe a similar trend to the TX case where the ACP bandwidth is higher when most of the data are cached.

B. Software Cost Evaluation

In this section, we evaluate non-cacheable memory access bandwidths and manual cache operation costs. The advantage of using caches is well evaluated in the past [2], [3], but we include the evaluation in this paper for the completeness of I/O cache coherency evaluation. For the non-cacheable memory access evaluation, we first measure four types of memory copy operations: non-cacheable to non-cacheable, non-cacheable to cacheable, cacheable to non-cacheable, and cacheable to cacheable. All memory copies are done using `memcpy()` function from the C library. In Fig. 4 (a), we find the bandwidth penalty is as large as 30× when reading from the non-cacheable region compared to reading from the cacheable region. On the other hand, the memory writes to the non-cacheable regions remains almost the same because the Write-Combine (WC) function can combine multiple non-cacheable write requests to a single larger memory write. This feature will be further discussed in Section V-A.

Still, the WC is only active in regular memory access patterns and CPUs can suffer from long memory latencies with irregular memory write patterns. In Fig. 4 (b), we measure execution times of matrix transpositions to different types of memory. In this experiment, the source matrix is stored in cacheable memory region and the destination for the transposed matrix is located in non-cacheable memory region. When the entire matrix can fit in the cache, the cacheable memory is about 4× faster than the non-cacheable memory. When the matrix size is much larger than the cache size, the cacheable memory is still about 1.33× faster than the non-cacheable memory.

When manual cache instructions are needed, the CPU overhead added heavily depends on other CPU workloads and the total number of buffers flushed or invalidated. In Linux, after each buffer is flushed or invalidated, global memory barrier should be inserted to guarantee no memory accesses are reordered. If this global memory barrier needs be executed multiple times while heavy memory accesses are being made, the overall CPU performance can be severely degraded. In our experiment, we found averagely 50% of execution time

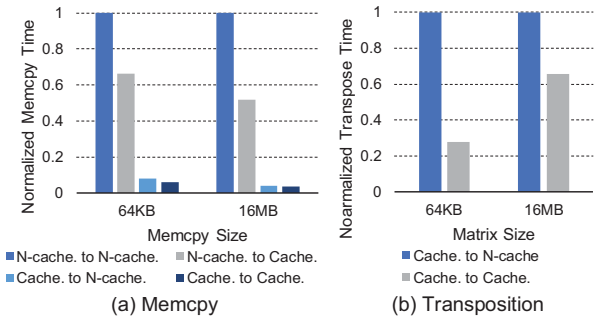

Fig. 4. (a) Memcpy execution time comparison using different combinations of cacheabilities. (b) Matrix transpose execution time comparison with non-cacheable and cacheable destination buffers.

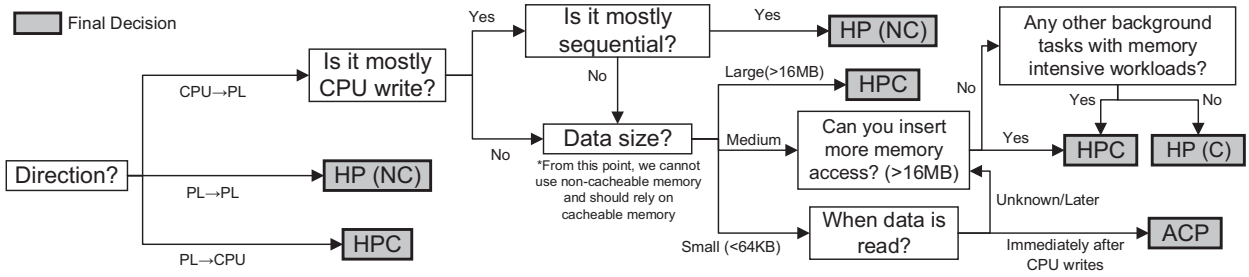

Fig. 5. Decision tree for selecting the optimal I/O cache coherence method.

is spent for the manual cache instructions regardless of the direction.

V. Optimizing Data Transactions

In this section, we suggest several I/O cache coherence optimization techniques to achieve the most effective data transaction behaviors. First, we introduce several hardware features which can be exploited to remove some cache coherence overheads. Second, we present a decision tree (Fig. 5) which can be utilized to optimize I/O design. Finally, we apply our decision tree to several applications and evaluate.

A. Exploiting Hardware Features

1) Wribe Combine (WC): WC is a cache feature which can combine multiple write accesses to non-cacheable regions into a single larger memory write request [4]. Compared to requesting multiple small memory writes, requesting a single larger memory write can better utilize the memory bandwidth. To activate this feature, the target addresses of consecutive write requests should be contiguous.

2) Cache Bypass: In Section IV-A, we showed the CPU→PL bandwidth of HPC interface can be significantly lower when the data is cached. It is possible to resolve this by manually flushing cache lines, but this costs CPU cycles in exchange. One way to implicitly flush the cache lines is using cache bypass function in hardware [5]. Cache bypass can be used in cacheable memory region where caches decide not to allocate certain cache lines for certain data access patterns. This kind of behavior can be often observed when using `memset()`. With this feature, without explicitly executing cache flush instructions, data can be directly written into DRAM even if the memory regions are cacheable.

B. I/O Cache Coherence Decision Tree

Gathering all explorations from previous sections, we build a decision tree (Fig. 5) to provide a general I/O cache coherence optimization flow. The total cost of I/O cache coherence can be roughly estimated as follows:

$$(total\ cost) = \frac{\alpha}{(raw\ bandwidth)} + (software\ cost)$$

Here, the α represents the bandwidth requirement of an application. We first categorize all data transaction types into CPU to PL, PL to PL, and PL to CPU. Just to clarify, in this decision tree, we are only accounting to the cases where a shared memory (mostly host DRAM) between two instances is used as a data communication medium. Our decision tree strategy focuses on minimizing unexpected risks rather than maximizing possible gains.

For the communication between PL logics, there is no CPU involvement and therefore using HP (NC) is the best. For the PL to CPU case, we conclude using HPC interface is the best in general as it can provide relatively high memory bandwidth while not introducing additional software costs. The memory bandwidth loss with the HPC interface in this case compared to the HP is only about 5% (Fig. 3).

CPU to PL case is more complex than the former two cases as the raw bandwidth differences are huge in this case. In this case, we first check if the TX buffer is mostly used for CPU write. If the CPU is mostly writing to the buffer, then we check if the writing is mostly done in sequential manner. If the memory write patterns are sequential or can be modified to be sequential, then we can safely use the non-cacheable memory allocation. If the writes cannot be made sequential or the CPU needs to make substantial amount of read requests from this buffer, the buffer cannot be made non-cacheable. From this point, we need to rely on HP (C), HPC, or ACP.

Using HP (C) is discouraged in general since executing extra cache instructions and memory barriers can only have negative affects in terms of performances. To use HPC or ACP, we must check how much of the data to be transferred is cached as the raw bandwidths of HPC and ACP vary a lot depending on the data locations. However, because it is impossible to know the exact location of data before we access the cache, we rely on several intellectual guesses. First, we check the size of the data. If the data size is large enough (>16MB), based on our observation from Fig. 3, we can obtain relatively high bandwidth with HPC. Second, if the data size is small (<64KB) and the accelerator reads the data immediately after the CPU writing, we can use ACP to maximize the bandwidth. Third, if none of the above cases were true, we can consider reordering some other workloads to just before the accelerator data reads. For example, in video streaming, we can add some delay of several frames to make latter frames to evict former frames from the cache. If the reordered workloads can make large enough amount of memory accesses (>16MB), most of the data will be evicted from the cache and we can use HPC. If this is also impossible, then we finally need to consider using HP (C). Before we choose HP (C), one thing we may

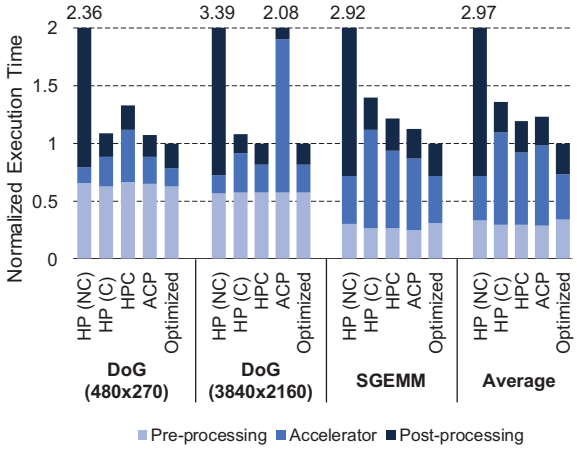

Fig. 6. Benchmark results with using different I/O cache coherence methods. Difference of Gaussian (DoG) is tested with different image sizes.

need to consider is if there are any background tasks which are memory intensive. If there are any such tasks, we should still consider using HPC as memory barriers inserted by HP (C) is likely to slow down the overall CPU performances.

C. Case-Study Evaluations

To evaluate our decision tree, we use modified *Difference of Gaussian (DoG)* filter from *xfOpenCV* [6], *SGEMM*, and *CHaiDNN* [7] with *AlexNet* as case-study examples. All applications are written in C++ and synthesized with Xilinx SDSoC. DoG takes grayscale images as inputs and generates two outputs. For this application, we use CPU to convert RGB images to grayscale images and subtract two gaussian filtered images. Accelerator is used for accelerating the gaussian filters. For SGEMM, we implement a 128×128 matrix multiplication accelerator and perform block matrix multiplication for larger input matrices. CPU is responsible of cropping input matrices into 128×128 blocks and feeding into the SGEMM accelerator and accumulating the accelerator outputs into the output matrix. CHaiDNN accelerates convolution and pooling layers of DNN and CPU is responsible of quantizing input images and de-quantizing accelerator outputs.

For the baselines, we implement designs with pure HP (NC), HP (C), HPC, or ACP options. Due to the design complexity, we only compare between HP (NC), HP (C), and optimized version for CHaiDNN. The optimized designs follow the decision tree we created.

Fig. 6 shows the benchmark results of DoG with different image sizes and SGEMM. In average, our optimized version achieved at least 20% of execution time reduction compared to any other baseline configurations. In general, HP (NC) has the smallest accelerator execution times due to is high raw bandwidth, but the post-processing times have been greatly increased. HP (C) in general has very long accelerator execution times because of manual cache instructions and memory barriers. HPC performs well when the input sizes are large, but starts to suffer from low raw bandwidth when the inputs are small due to the reason explained in Section IV-A. In opposite,

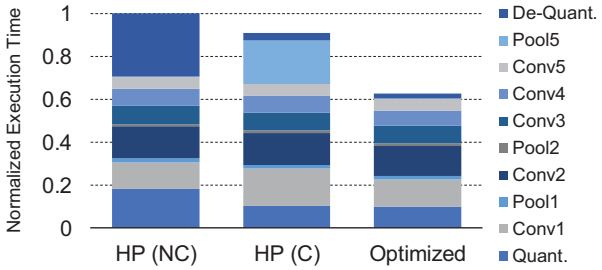

Fig. 7. Benchmark results of CHaiDNN with different I/O cache coherence methods. Quantizations and de-quantizations are done at CPU. The execution order is from bottom to top (Quant.→Conv1→...→Pool5→De-Quant.).

ACP performs well when the input sizes are small, but as the input sizes increase the cache hit rates become lower and the accelerator execution times start to skyrocket.

Fig. 7 shows the AlexNet execution time breakdown with CHaiDNN. HP (NC) greatly suffers from non-cacheable memory accesses during both quantizations and de-quantizations. HP (C) has slightly better performance than HP (NC), but still need to spend non-negligible amount of time executing manual cache instructions. The optimized version removes the penalties of both HP (NC) and HP (C) and reduces the execution time by 37.2% and 30.9% compared to HP (NC) and HP (C), respectively.

VI. Related Works

There are several I/O cache coherence bandwidth researches with older SoC-FPGA platforms such as Xilinx's Zynq-7000 and Altera's Cyclone V [8]–[14]. For both platforms, the only available hardware coherent I/O port is ACP. [9]–[14] are limited to evaluating raw I/O bandwidths of using different ports and did not include software cost evaluations. [8] has evaluated software costs of I/O cache coherence, but only with a fixed data access pattern.

VII. Conclusion

The costs of different I/O cache coherence methods varies widely depending on applications. Approaching the I/O cache coherence optimization problem should be done in bottom-up fashion including both software and hardware profilings. In this paper, we presented multiple I/O cache coherence methods of SoC-FPGA and optimization techniques based on thorough analysis of Zynq UltraScale+ platform. By properly combining different I/O cache coherence methods, we showed the overall execution time can be reduced by 20%. In this paper, we mainly discussed the I/O cache coherence in a context of CPU-to-accelerator connections, but this can be also applied to other device connections such as high-speed Ethernet, GPU, and NVMe.

VIII. Acknowledgements

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, and IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) – a research collaboration as part of the IBM AI Horizon Network.

REFERENCES

[1] T. B. Berg, "Maintaining I/O data coherence in embedded multicore systems," *IEEE micro*, vol. 29, no. 3, 2009.

[2] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 364–373.

[3] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 63–74.

[4] J. Benkual, T. Y. Ho, and J. F. Duluk Jr, "System, apparatus, method, and computer program for execution-order preserving uncached write combine operation," Dec. 30 2003, uS Patent 6,671,747.

[5] T. L. Johnson, D. A. Connors, M. C. Merten, and W.-M. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[6] Xilinx, "xfOpenCV," https://github.com/Xilinx/xfopencv, 2019.

[7] ——, "CHaiDNN," https://github.com/Xilinx/CHaiDNN, 2019.

[8] A. Powell and D. Silage, "Statistical performance of the ARM cortex A9 accelerator coherency port in the xilinx zynq SoC for real-time applications," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–6.

[9] J. Silva, V. Sklyarov, and I. Skliarova, "Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31–34, 2015.

[10] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ," in *Proceedings of the 10th FPGAworld Conference*. ACM, 2013, p. 5.

[11] P. Vogel, A. Marongiu, and L. Benini, "An evaluation of memory sharing performance for heterogeneous embedded SoCs with many-core accelerators," in *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*. ACM, 2015, p. 6.

[12] V. Sklyarov, I. Skliarova, J. Silva, and A. Sudnitson, "Analysis and comparison of attainable hardware acceleration in all programmable systems-on-chip," in *2015 Euromicro Conference on Digital System Design*. IEEE, 2015, pp. 345–352.

[13] R. F. Molanes, J. J. Rodríguez-Andina, and J. Farina, "Performance characterization and design guidelines for efficient processor–FPGA communication in Cyclone V FPSoCs," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 5, pp. 4368–4377, 2018.

[14] R. F. Molanes, F. Salgado, J. Fariña, and J. J. Rodríguez-Andina, "Characterization of FPGA-master ARM communication delays in Cyclone V devices," in *IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2015, pp. 004 229–004 234.