Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ

Mohammadsadegh Sadri*, Christian Weis†, Norbert Wehn†, and Luca Benini*
*Department of Electrical, Electronic and Information Engineering (DEI) University of Bologna, Italy
†Microelectronic Systems Design Research Group, University of Kaiserslautern, Germany
{mohammadsadegh.sadr2,luca.benini}@unibo.it, {weis,wehn}@eit.uni-kl.de

ABSTRACT

Cooperation of CPU and hardware accelerator to accomplish computational intensive tasks, provides significant advantages in run-time speed and energy. Efficient management of data sharing among multiple computational kernels can rapidly turn into a complicated problem. The Accelerator coherency port (ACP) emerges as a possible solution by enabling hardware accelerators to issue coherent accesses to the memory space. In this paper, we quantify the advantages of using ACP over the traditional method of sharing data on the DRAM. We select the Xilinx ZYNQ as target and develop an infrastructure to stress the ACP and high-performance (HP) AXI interfaces of the ZYNQ device. Hardware accelerators on both of HP and ACP AXI interfaces reach full duplex data processing bandwidth of over 1.6 GBytes/s running at 125 MHz on a *XC7Z020-1C* device. The effect of background DRAM and cache traffic on the performance of accelerators is analyzed. For a sample image filtering task, the cooperative operation of CPU and ACP accelerator (*CPU-ACP*) gains a speed-up of $1.2X$ over CPU and HP acceleration (*CPU-HP*). In terms of energy efficiency, an improvement of 2.5 nJ ($> 20\%$) is shown for each byte of processed data. This is the first work which represents detailed practical comparisons on the speed and energy efficiency of various processor-accelerator memory sharing techniques in a configurable heterogeneous platform.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*

General Terms

Design, Performance, Measurement

Keywords

Zyqn, Accelerator Coherency Port, AXI masters

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGAWorld '13, September 10-12, Copenhagen, and Stockholm.
Copyright 2013 ACM 978-1-4503-2496-0/13/09 ...$15.00.

1. INTRODUCTION

As the energy efficiency requirements (e.g. GOPS/W) of silicon chips are growing exponentially, computer architects are seeking solutions to continue application performance scaling. One emerging solution is to use specialized functional units (accelerators) at different levels of a heterogeneous architecture. These specialized units cannot be used as general-purpose compute engines. However, they provide enhanced execution speed and power efficiency for their specific computational workloads [3]. There exist numerous applications for accelarators in both of the embedded and high performance computing markets. Examples include video processing [24], software-defined radio [5], network traffic management [19], DNA computing [17] and fully programmable hardware acceleration platforms [23]. Efficient sharing of data in a heterogeneous MpSoC which contains different types of integrated computational elements is a challenging task. Especially when private caches of CPU cores and dedicated memory of accelerators are used to store local copies of data, it is crucial to ensure that every processing element has a consistent view of the shared memory space [28], [18].

The Accelerated coherency port (*ACP*) [27] was developed by ARM® as a hardware solution to facilitate dealing with cache coherency issues when introducing new accelerator blocks to a multi-core system. In fact, *ACP* enables hardware accelerators to issue coherent requests to the CPU sub-system memory space [10].

Xilinx ZYNQ all-programmable SoC [16] provides the designers with an ARM *Cortex-A9 MPCore* sub-system along with a high performance DRAM memory controller [12] and various peripherals. It also implements a complete FPGA fabric. The CPU sub-system and FPGA are connected through AXI [11] interfaces which allow the logic on the fabric to perform cache coherent accesses to the memory through *ACP* or directly perform accesses to the DRAM. Section 1.2 further describes the *ZYNQ* architecture.

In this paper, we use the ZYNQ device and build a complete infrastructure to evaluate the performance and energy efficiency of different processor-accelerator memory sharing schemes. We pass through the following steps:

- Setup of an infrastructure containing hardware (section 2.1) and required kernel and user level software (section 2.2).
- Define a computational task and also the set of processor-accelerator memory sharing methods used to perform this task (section 3).

- Evaluation of speed and energy efficiency of each method by running practical experiments on the hardware (section 4).

- Description of the lessons learnt from the results.

The authors provide the infrastructure, containing hardware projects, firmwares and developed software as open source code and free of charge to the research community.

1.1 Related Work

Heterogeneous System Architecture (HSA) foundation provides architectural and application level solutions to help system designers integrate different kinds of heterogeneous computing units in a way that eliminates the inefficiencies of sharing data and sending work items between them [20]. The developed acceleration hardware blocks become HSA compliant by declaration of the necessary low-level interface layers. This frees the programmers from the burden of tailoring a program to a specific hardware platform. As a part of HSA, [25] describes *AMD Fusion System Architecture*; targeted to unify CPUs and GPUs in a flexible computing fabric. The proposed idea however, is mainly developed for sharing memory between CPU and fully programmable GPU cores and is not targeting reconfigurable heterogeneous architectures like *ZYNQ*.

A methodology for analyzing the impact of hardware accelerator data transfer granularity on the performance of a typical embedded system is presented in [21]. This is particularly important because, as we will show, the granularity of data transfer between memory and accelerator, and thus, the interrupt rate to the CPU has direct impact on the performance.

The idea of using a portion of CPU sub-system caches as buffers for the accelerators is studied in [6]. This results in smaller silicon area since each accelerator doesn't instantiate its own buffer. The basic idea of dedicating a shared memory space to accelerators is interesting because the *ZYNQ* device provides a dedicated On-Chip Memory (*OCM*) which can be used for the same purpose. We also consider processor-accelerator memory sharing using *OCM* in our tests.

The problem of maintaining coherency between CPU caches and accelerator data in a multi-core embedded system is addressed in [2]. The paper discusses possible hardware architectures and related software solutions to tackle the problem. It concludes that the optimal solution heavily depends on the characteristics of the application. The paper discusses the solutions at architecture level and does not provide detailed practical comparisons on the performance and energy efficiency of each solution.

An area- and power-efficient many-core computing fabric which features clusters of up to 16 processor cores is proposed in [1]. The developed platform delivers an extraordinary level of computational speed ($>$ 80 GOPS) while consuming relatively small amount of power ($<$ 2 W). The paper is of particular importance since it provides ideas on the development of energy efficient accelerator logic. Indeed, the developed architecture in our paper is partially inspired from the method used by [1] to connect to its host CPU.

A high-performance, energy improved mobile processing platform named big.LITTLE is introduced by [7]. The platform consists of high performance Cortex-A15 processor and energy efficient Cortex-A7. The connection between the CPU sub-systems is provided through the CCI-400 interconnect which facilitates full coherency between Cortex-A15 and Cortex-A7 as well as GPUs, accelerators and I/O. This platform, if connected to a programmable gate array, can provide a suitable testbed for evaluation of various processor-accelerator memory sharing schemes.

The impact of cache architecture on the performance and area of FPGA based processor and parallel accelerator systems is discussed in [4]. The paper proposes a simple hardware containing one MIPS core, multiple accelerator units, a multi-port shared L1 cache and a DRAM controller. It considers different structural parameters for the L1 cache (such as number of ports, associativity, etc.) and defines a set of computational tasks to be done only by accelerators. It then quantifies the impact of cache structure on the overall speed of accelerators connected to the L1 cache. The paper does not discuss the cooperative operation of CPU and accelerators. Moreover, the developed hardware in the paper is very simple. It is not capable of booting an operating system and communicating with the outside world.

The idea of adding hardware accelerators to reduce power in FPGAs is investigated in [8]. The paper shows practical comparisons for the power consumption of sample computational tasks, when they are executed by the CPU or the accelerator logic. The paper does not address issues related to coherency and processor-accelerator memory sharing.

To the authors' knowledge, this is the first work which practically quantifies the potential processing bandwidth and energy efficiency of different processor-accelerator memory sharing methods using the *ZYNQ* device.

Moreover, it is the first work which provides an explicit practical comparison in terms of energy and speed, on processor-accelerator memory sharing using *ACP* and other traditional methods. In addition, we also provide a flexible research vehicle which facilitates evaluation of innovative ideas regarding the design of hardware accelerators in heterogeneous architectures on programmable gate arrays.

1.2 Key ZYNQ Architecture Description

Xilinx ZYNQ device [16] contains two parts: 1-Programmable Logic *(PL)* which is roughly a full FPGA. 2-Programmable System *(PS)* which is a complete sub-system with ARM CPU cores and different peripherals [26]. A set of AXI interfaces (as shown in Figure 1) are implemented to make the communication between PS and the PL logic possible. Basically these AXI interfaces divide into two groups:

- AXI Master interfaces (*GP*), connect to AXI slaves residing on the PL. The CPU is able to initiate read/write transactions over these AXI masters to transfer data to PL modules. There are two 32 Bits AXI master ports available in the ZYNQ device: *GP0* and *GP1*.

- AXI Slave interfaces (*HP*, *ACP* and *SGP*), connect to the implemented AXI masters on the PL. There exist four *High Performance* (HP) ports and one *Accelerator Coherency Port* (ACP). Each of these interfaces implements a full-duplex 64 Bits connection, meaning that at every clock cycle, total 16 Bytes of data can be transferred on *AXI read* and *AXI write* channels concurrently. The two *SGP0* and *SGP1* interfaces implement 32 Bits connections.

There exists a defined memory map for the *ZYNQ* device [16] which indicates the address range of each logic block. Every

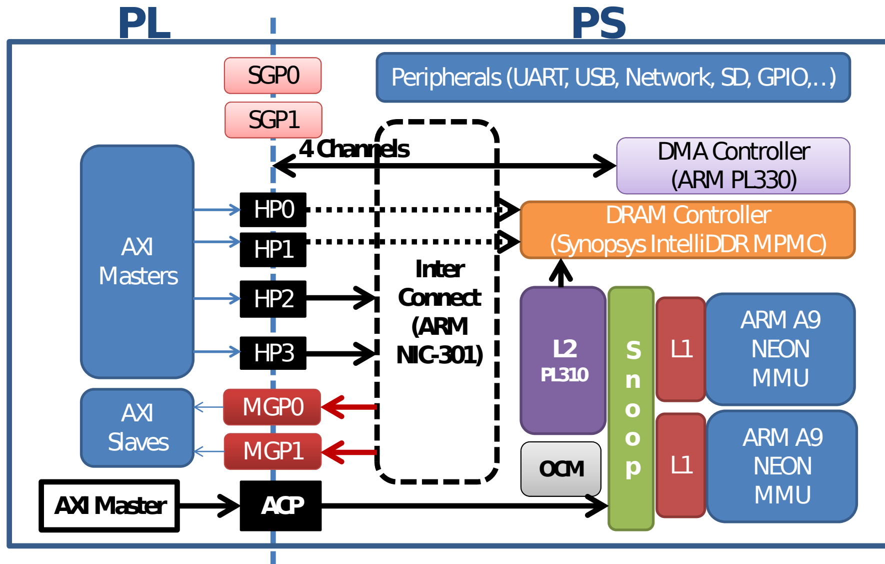

Figure 1: A block diagram representing important elements of the Xilinx ZYNQ device.

AXI slave unit, implemented on the PL will also occupy a part of this address range. It should be noted that except the CPU cores and their $L1$ instruction caches, the rest of the system is using physical address values.

The ACP port is connected to the ARM Snoop Control Unit (SCU).Thus it provides the possibility of initiating cache coherent accesses to the ARM sub-system. Careful use of ACP can improve overall system performance and energy efficiency. Inappropriate usage of this port however, can adversely affect execution speed of other running applications because the accelerator can pollute precious cache area.

2. INFRASTRUCTURE SETUP

We develop a complete infrastructure containing hardware, software and firmware setup which enables us to perform evaluations on different processor-accelerator memory sharing methods. For the firmware, we basically use the generated firmware by Xilinx toolset for our target board (*ZC-702*). We ensure that AXI level shifters are enabled from the begining of device operation. This is vital for the correct operation of HP and ACP interfaces.

2.1 Hardware

Figure 2 shows a block diagram of the developed hardware on the ZYNQ device. As we see, three AXI slave interfaces (*ACP*, *HP0* and *HP1*) and one AXI master interface (*GP0*) are enabled and used. The AXI masters used in this design implement *AXI 4.0* protocol specifications [11]. They are based on the AXI master template provided as a LogiCORE by Xilinx [13]. Each AXI master logic is further customized to issue an interrupt when it finishes a transfer task. The interrupt signals are connected to the interrupt controller unit on the PS. Each AXI master, also contains an AXI slave port, through which the CPU can program and start the master. All of the AXI masters are configured to handle burst lengths of up to 256 which is equal to 4096 Bytes of data (*read+write*). Each AXI master, when programmed, is capable of handling transactions of up to 1 MBytes total length. The AXI slave side of all of the AXI master units residing on *ACP*, *HP0* and *HP1* ports are connected to *GP0*.

In order to push the processing speed to its maximum, we use two AXI master blocks (called *AXI read* and *AXI write*) running in parallel to perform concurrent read and write transactions on each PS interface. The masters are connected to the interface using an AXI interconnect. In Figure 2 each AXI interconnect is depicted with a big i letter. AXI interconnects are configured to their high performance cross-bar mode to achieve maximum possible bandwidth.

As shown in Figure 2, between *AXI read* and *AXI write* there is a separate module (called *acceleration logic*) which contains a FIFO and also the logic related to the acceleration task. For our performance measurements, we have selected a 16 tap FIR filter as the acceleration logic. Each pair of AXI masters can operate in parallel, meaning that, while one of them is reading a packet of data from the memory, the other one is writing the previous processed packet back to its destination. The developed driver at software side is resposible for synchronization of these two units.

The proposed hardware provides the designers with an architecture which is very easy to customize. For any defined computational task, the *acceleration logic*, which is based on an easy to understand FIFO interface, is the only block which needs to be modified.

The hardware also includes an AXI monitor unit [14]. Its purpose is to be able to monitor the AXI interface signals, so that we can debug possible problems and further investigate latency values by directly looking at the actual waveforms.

If the AXI interfaces are running at 125 MHz the maximum theoretical full-duplex bandwidth for them is 2 GBytes/s. For DRAM memory, the data bus width is 32 Bits and if *DDR3 DRAM* is running at 533 MHz, we reach a theoretical bandwidth of 4.2 GBytes/s.

Finally, we have also placed a set of AXI masters on the *HP1* port. The purpose of these blocks is to be able to generate dummy traffic to the DRAM whenever required. Using this block we will evaluate the performance of the accelerator running on *HP0* while the DRAM is also busy

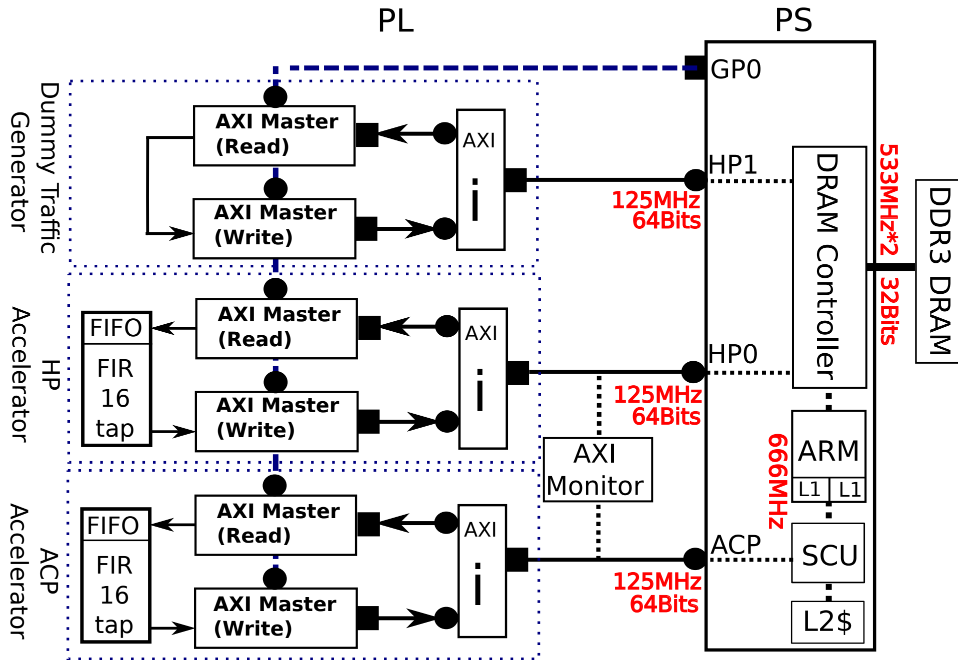

Figure 2: Block diagram of the developed hardware.

handling other incoming requests.

We have implemented this hardware on the ZYNQ *XC7Z020-1C* device available on Xilinx *ZC-702* board. The total number of used logic slices is equal to 7324 which corresponds to 55% of the total available slices. The design consumes 92 block memories of 36 Kbits size (65%) and 9 block memories of 18 KBits size (3%). The maximum clock frequency for this design is 128.2 MHz.

2.2 Software Environment

Our developed software is divided into two major parts: Linux kernel level drivers and user level applications.

At Linux kernel level, our infrastructure consists of two drivers which are called *axiD* and *axiD dummy generator*.

axiD manages the AXI masters located on *HP0* and *ACP* ports. The driver is responsible for:

- Memory allocation and obtaining the physical address of allocated memory buffers which will be used by AXI masters. Memory can be allocated in either cachable or non-cachable regions depending on the memory sharing method. (Further descriptions in section 3.)
- Initializing, programming and triggering the AXI masters and handling the interrupts generated by them.
- Calculating the source and destination addresses for the set of accelerators based on the status of the ongoing processing tasks, number of passed loops and number of processed data chunks.
- Interaction with user-level applications: receiving raw input data from user side, copying to source memory buffers and then writing back the processed results to user-level.
- Providing an accurate tool to measure time intervals. The driver enables access to the free running $64 Bits$ counters of the ZYNQ device [16] which are clocked at $333 MHz$ (half CPU clock frequency).
- Configuring the $PL310$ [10] cache controller statistics unit so that it reflects total number of read requests received at the cache and the total number of read hits. The driver reflects these values at the beginning and ending time of each task.

The developed *axiD dummy generator*, does not perform any acceleration related task. When needed, it enables us to activate the dummy traffic generator AXI masters residing on *HP1* port.

At user level, we have prepared the following items:

- A simple application which communicates with the *axiD* driver.
- A simple memory intensive application (called *background application*), which allocates a memory buffer and performs arbitrary read and writes to this buffer in an endless loop. This application will be used to demonstrate the effect of cache pollution on the performance of *ACP* accelerator.
- The *Oprofile* statistical performance monitoring tool [22] which we have ported to the ZYNQ environment. This enables us to measure important performance metrics of the CPU sub-system.

2.3 Power Measurement

The *ZC-702* board is utilizing a set of power supply units which provide online sensors for voltage, current and temperature measurement [15]. Basically we sample the following consumed power values: 1-Core logic for the *PL* part of *ZYNQ*. 2-Internal logic of the *PS*. 3-Interfaces and I/O buffers of the *PS* and 4-DRAM chips. Based on our practical observations, these four items are the most power hungry

parts of the system. The sampling frequency for measurement of voltage and current values is equal to 2 Hz.

3. MEMORY SHARING METHODS

In order to evaluate different processor-accelerator memory sharing methods in terms of speed and energy efficiency, we first define a processing task. Then we define a set of processing methods to accomplish this task. Each processing method utilizes a different memory sharing scheme. We then execute each processing method on the real hardware and measure performance and power.

Processing Task: For a sample image of i bytes, perform the following: read the image from the source buffer, pass the image through the FIR filter, and finally write the output back to the destination buffer. Source and destination buffers are different. In practice, we continuously perform this operation a large number of times. This enables us to have an accurate speed and power measurement.

Processing Methods: Here, we describe the methods that we use to perform the processing task. We assign a name to each method, which will be used during the rest of this paper.

- *HP0 Only*: The accelerator located on *HP0* is responsible to perform the processing task alone. Image source and destination buffers are allocated on the DRAM memory and in the non-cachable area. (Linux kernel call *dma_alloc_coherent* is used for this purpose.)
- *ACP Only*: The accelerator located on *ACP* is responsible to accomplish the processing task alone. Image source and destination buffers are allocated using normal *kmalloc* Linux kernel call, thus, they are allowed to also be cached by CPU sub-system.
- *OCM Only*: The accelerator located on *ACP* is reponsible to accomplish the processing task alone. However, image source and destination buffers are located in the On-Chip Memory (*OCM*) block of the ZYNQ device. Here the allocation will be done like other hardware peripherals using *request_mem_region* and then *ioremap* Linux kernel calls.
- *CPU Cache*: The CPU core is reponsible for doing the processing task alone. No accelerator is active. The source and destination image buffers are allocated using *kmalloc* thus, they are allowed to get cached.
- *CPU no Cache*: Is similar to *CPU Cache* however, memory allocation for the source and destination buffers is done using *dma_alloc_coherent* thus they are located on non-cachable region of memory.
- *CPU HP0*: The CPU and the accelerator on *HP0* port cooperate to perform the processing task. At each iteration first the CPU reads the source image, performs the processing and writes the result back to the memory. Then it is the turn of the accelerator on *HP* port to perform the processing task. Image source and destination buffers are allocated on non-cachable region of memory.
- *CPU ACP*: Is similar to *CPU HP0* however, the accelerator on ACP cooperates with CPU to accomplish the task and image buffers are allowed to be cached.

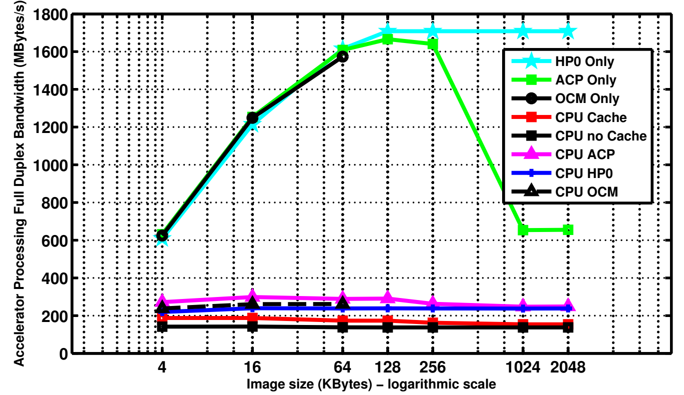

Figure 3: Processing bandwidth comparison of acceleration methods. Image size sweeps from 4 KB to 2048 KB.

- *CPU OCM*: Is similar to *CPU ACP* however, source and destination image buffers are located on the *OCM*.

4. EXPERIMENTAL RESULTS

We consider the processing task described in section 3 and we use each of the described methods to accomplish this task and to measure processing speed and energy.

We sweep over different image size i values to evaluate the effect of used memory size on the speed of operation. ($i = \{4, 16, 64, 128, 256, 1024, 2048\}$ KBytes). By increasing i, we also increase the size of packets (p) transferred by the AXI masters ($p = \{4, 16, 64, 128, 128, 128, 128\}$ KBytes). Although our AXI masters are capable of handling packets of up to 1 MBytes, we limit the packet size to 128 KB which is the size of FIFOs inside *acceleration logic*. During these tests, we use a fixed running frequency of 125 MHz for the entire logic residing on the *PL*. We measure total execution time and thus total processing bandwidth (*read+write*) for each case. During each test we also measure total number of $L2$ cache requests and hits to have a better insight on $L2$ cache utilization. Figure 3 shows the total processing bandwidth for each method. The Y axis represents total transferred data ($read + write$) in MB/s. X axis represents the size of image (i) being processed in a logarithmic scale.

The following processing methods show highest performance: *HP0 Only*, *ACP Only* and *OCM Only*. For *OCM Only* we can perform the processing task only for limited values of i since, the total On-chip Memory available on the *ZYNQ* device is limited to 256 KB. For $i = \{4, 16, 64\}$ KB we see almost equal performance for each of these three methods. At $i = 128$ KB and $i = 256$ KB, we notice a slight decrease in the performance of *ACP Only* compared to *HP0 Only* (1708.5 MB/s for *HP0 Only* vs. 1665.9 MB/s and 1640.8 MB/s for *ACP Only*). When image size grows over 256 KB, a significant drop appears in the performance of *ACP Only* (653.3 MB/s for *ACP Only* vs. 1708.5 MB/s for *HP0 Only*).

This phenomena can be described as follows; For each processing task, the total utilized memory (by source and destination image arrays) is $2 \times i$. If the total available cache size is L, then while $2 \times i < L$ the system is able to efficiently store local copies of recently used *ACP* accelerator data on its caches, thus providing fast access to data when

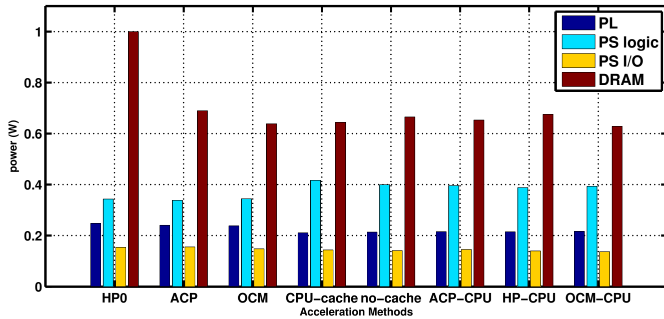

Figure 4: Averaged power consumption of major system blocks for each processing method.

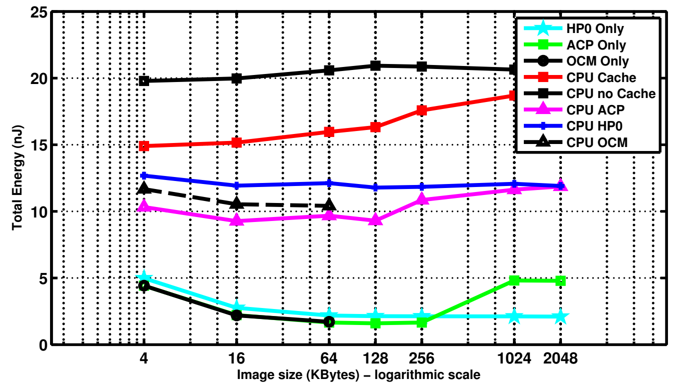

Figure 5: Energy consumed for processing of one byte of data for each processing method.

needed. However when $2 \times i > L$, it is no more possible to cache the entire data objects used by the accelerator. As a result, some accelerator requests to the cache will fail and will eventually end-up the DRAM memory. The extra delay introduced by passing through the caches to DRAM, causes a serious decrease in performance. Either an increase in i (e.g. increasing the size of processed image) or a decrease in L (e.g. a background application is also consuming available caches) can cause the above phenomena. In Figure 3, no background application is running on the system. The size of available shared $L2$ cache is 512 KB in the ZYNQ device. As we see, performance drop happens when $2 \times i > 512$ KB.

We now consider processing methods which fully or partially use the CPU cores to perform the processing task. *CPU no cache* method is showing the lowest performance (average 140 MB/s) and *CPU cache* is slightly higher (average 170 MB/s). Here, the entire processing is done only with the ALU of the CPU. Enhancements in speed is possible if we use the *NEON SIMD* engine of ARM CPU cores. But even in that case the possible speed-up is around $8X$ [9].

Finally, we have *CPU ACP*, *CPU OCM* and *CPU HP0* with speeds between CPU only and hardware only methods. Here, cooperation of accelerator with the CPU causes an speed-up in data processing. *CPU ACP* is always faster than *CPU HP0*. This is because of the possibility of sharing the data between CPU and accelerator on the cache (Thus the CPU can access data faster). The speed of *CPU OCM* is always between the other two methods. For $i \leq 256$ KB *CPU ACP* is approximately $1.22X$ faster than *CPU HP0*. By growing the image size however, the speed of *CPU ACP* begins converging to *CPU HP*.

Looking at the number of $L2$ cache hits, we notice a significant difference between the methods which use *ACP* and methods which use *HP0*. For example, in *ACP Only* we see more than 2 hits per each 32 bytes (one cache-line) of processed data while for *HP0 Only* this value is practically zero (in the order of 10^{-5}).

For each test point in Figure 3, we also measure power. Figure 4 shows the results. Considering the fact that, power values do not change significantly by changing the image size for each processing method, we only show the averaged power value for all of the image sizes. In Figure 4 we show the four major power sinks (as described in section 2.3) of the *ZC-702* board at the time of the tests. As shown, *HP0 Only* method has the highest DRAM power. *CPU cache* causes highest power consumption by PS internal logic, and *HP0 Only*, *ACP Only* and *OCM Only* show highest values of power consumption by the PL. Power consumption of *PS* I/O buffers are at the same level for all methods. Having the processing bandwidth and power, we calculate the energy consumed for processing one byte of data. Figure 5 shows energy values for each of the test points.

Looking at Figure 5 we see, for $i \leq 256$ KB, *ACP Only* and *OCM Only* consume the least energy. For $i > 256$ KB however, *HP0 Only* shows better results. At the next level, among methods which utilize the CPU, cooperation of CPU and accelerator over ACP (*CPU ACP*) shows the lowest energy. After that, we have *CPU OCM* and then *CPU HP0* showing more energy consumption.

4.1 Effect of Background Workloads

Now, we study the effect of background workloads on the performance of *ACP Only* and *HP0 Only* methods. First, we turn on the dummy traffic generator on *HP1* AXI interface. This block continuously performs arbitrary read and write operations to an allocated 2 MB DRAM area. At the same time we measure the performance of *ACP Only* and *HP0 Only* for different values of i.

In another test, we execute a memory intensive *background application* on ARM CPU cores. The dummy AXI traffic generator is off. The *background application* performs arbitrary read and write operations to an allocated 2 MB array. The array is allowed to be cached. Thus, it occupies CPU caches during execution. In fact, this test shows the effect of decreasing L on the performance of the acceleration method.

Figure 6 shows the results of both tests. In this figure, X axis is i and Y axis is total transferred data in MBytes/s. We first look at the effect of *AXI dummy* activity on performance. As we see, performance drop of (*ACP Only*) is negligible. However, a major drop can be seen in the performance of *HP0 Only*. For example, at $i = 128$ KB, *HP0 Only* speed is 1708.5 MB/s when there is no other activity going on DRAM. However, its speed drops to 1382.2 MB/s when the *AXI dummy* interface is active and occupying DRAM bandwidth. For *ACP Only* the corresponding numbers are 1665.9 MB/s and 1664.3 MB/s respectively.

Looking at the results of the second test where the cache is heavily occupied by the *background application*, we see a clear drop in the performance of *ACP Only* while *HP0 Only* shows a slight performance shift. In *ACP + background*

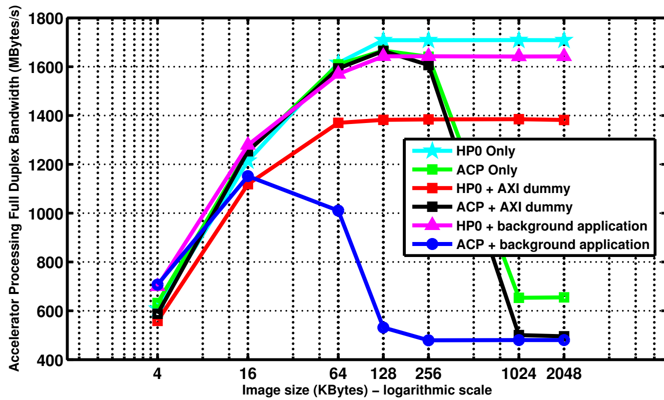

Figure 6: Processing bandwidth comparison of ACP and HP0 accelerators at the presence of background traffic.

application, the speed of the ACP accelerator does not grow for $i > 16$ KB. For example at $i = 128$ KB, the speed of *ACP Only* is 1665.9 MB/s and 531.6 MB/s with and without the *background application* running, respectively.

Another noticeable point during the second test is that: the operation speed of both *ACP* and *HP0* accelerators, at $i = 4$ KB, is higher when the *background application* is running on the CPUs compared to when the CPUs are not doing any specific task [1]. We describe this phenomena as follows: when the background application is running on CPU cores, it keeps the CPU sub-system active preventing it to go to idle state. Thus when an AXI master finishes its current task and issues an interrupt request to the CPU, the service routine will get executed in a shorter time, and the master begins the next task faster. This description can be further confirmed by noting the fact that when the packet size increases (and thus the rate of AXI master interrupts to the CPU decreases) this speed-up disappears. For further evaluation results please refer to [26].

5. LESSONS LEARNED

Based on the obtained results we derive the following design rules:

- If a specific task should be done by cooperation of CPU and accelerator: The speed and energy consumption of *CPU ACP* and *CPU OCM* methods are always better than (or in the worst case equal to) *CPU HP*.
  The main drawback of using *CPU ACP* is that parts of the available cache space will be occupied by the acceleration task. Thus if there exists any other critical application whose performance is heavily dependent on CPU caches, it may face problems handling its duty on-time. In this case, using *CPU OCM* (for small array sizes) and *CPU HP* (for big arrays) is recommended.

- If the task should be done by the hardware accelerator only (and then the CPU will just use the final result), then *ACP Only* or *ACP OCM* might be used only when the processed array blocks are small (smaller than the size of available cache, or on-chip memory) and there is no other background application consuming these resources. Otherwise, *HP Only* always provides better results.

[1] 700.0 MB/s for *HP0* and 707.3 MB/s for *ACP* when *background application* is running compared to 608.5 MB/s for *HP0* and 631.8 MB/s for *ACP* while the CPUs are idle.

The above rules can also be expanded to other platforms with similar architectures as the *ZYNQ*.

The *size of packets* transferred by each AXI interface, and *maximum burst length*, heavily affect the overall data transfer bandwidth. As a result, based on the traffic pattern of the processing task (e.g. the size of data chunks processed in each iteration), and provided implementation for AXI masters, interconnects and interfaces, these parameters should be set to the largest possible values.

6. CONCLUSION

In this paper, we demonstrated an assessment on the performance of acceleration by hardware blocks which communicate to CPU sub-system and DRAM over AXI interfaces. We selected Xilinx *ZYNQ* APSoC as the target and developed an infrastructure to measure the processing speed and energy. We compared the results when each of the CPU and accelerator perform the task alone and when the CPU and accelerator cooperate to accomplish the task. Based on the results, we derived a set of rules which can be used for efficient processor-acceleration memory sharing.

7. ACKNOWLEDGMENTS

This work was supported, in parts, by EU FP7 Project Virtical (GA n. 288574).

8. REFERENCES

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, 2012.

[2] T. Berg. Maintaining i/o data coherence in embedded multicore systems. *Micro, IEEE*, 29(3):10–19, 2009.

[3] C. Cascaval, S. Chatterjee, H. Franke, K. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5:1–5:10, 2010.

[4] 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*, pages 17–24, 2012.

[5] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. Magali: A network-on-chip based multi-core system-on-chip for mimo 4g sdr. In *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*, pages 74–77, 2010.

[6] C. Fajardo, Z. Fang, R. Iyer, G. Garcia, S. E. Lee, and L. Zhao. Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 966–971, 2011.

[7] P. Greenhalgh. big.little processing with arm cortex-a15 & cortex-a7. september 2011.

[8] Altera. Inc. Adding hardware accelerators to reduce power in embedded systems. september 2009.

[9] ARM. Inc. Introducing neon development, 2009. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/BABCJFDG.html.

[10] ARM. Inc. *Cortex-A9 MPCore Technical Reference Manual*, 2012. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0460c/CIAIIJCE.html.

[11] ARM. Inc. *AMBA AXI and ACE Protocol Specification*, February 2013. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html.

[12] Synopsys. Inc. *DesignWare DDR3/2 SDRAM Memory Controller*, 2013. http://www.synopsys.com/dw/ipdir.php?ds=dwc_ddr3_mem.

[13] Xilinx. Inc. *LogiCORE IP AXI Master Burst (DS844)*, June 2011. http://www.xilinx.com/support/documentation/ip_documentation/axi_master_burst/v1_00_a/ds844_axi_master_burst.pdf.

[14] Xilinx. Inc. *LogiCORE IP ChipScope AXI Monitor (DS810)*, March 2011. http://www.xilinx.com/support/documentation/ip_documentation/chipscope_axi_monitor/v2_00_a/ds810_chipscope_axi_monitor.pdf.

[15] Xilinx. Inc. *ZC-702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC*, April 2013. http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf.

[16] Xilinx. Inc. *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, March 2013. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[17] S. Ishikawa, A. Tanaka, and T. Miyazaki. Hardware accelerator for blast. In *Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium on*, pages 16–22, 2012.

[18] S. Kaxiras and A. Ros. Efficient, snoopless, system-on-chip coherence. In *SOC Conference (SOCC), 2012 IEEE International*, pages 230–235, 2012.

[19] A. Kennedy, X. Wang, and B. Liu. Energy efficient packet classification hardware accelerator. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[20] G. Kyriazis. Heterogeneous system architecture: A technical review. Technical report, Advanced Micro Devices, August 2012.

[21] S. Lafond and J. Lilius. Interrupt costs in embedded system with short latency hardware accelerators. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 317–325, 2008.

[22] J. Levon, M. Johnson, et al. Oprofile: A system profiler for linux. "http://oprofile.sourceforge.net/.

[23] O. Mencer. Maximum performance computing for exascale applications. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages iii–iii, 2012.

[24] M. Nadeem, S. Wong, G. Kuzmanov, and A. Shabbir. A high-throughput, area-efficient hardware accelerator for adaptive deblocking filter in h.264/avc. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 18–27, 2009.

[25] M. O'Connor. Accelerated processing and the fusion system architecture. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 93–93, 2012.

[26] M. Sadri. Technical report: Energy and performance exploration of accelerator coherency port using xilinx zynq. Technical report, Department of Electrical, Electronic and Information Engineering, University of Bologna, May 2013.

[27] N. C. Stephane Eric Sebastien Brochier. Managing the storage of data in coherent data stores, 09 2009.

[28] T. Suh, D. Blough, and H.-H. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1150–1155 Vol.2, 2004.