MASTER'S THESIS | LUND UNIVERSITY 2016

Image Processing Across Multiple Interconnected System-on-Chips

Andrée Ekroth, Felix Mulder

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-11

Image Processing Across Multiple Interconnected System-on-Chips

Andrée Ekroth
`dat11aek@student.lu.se`

Felix Mulder
`dat11fmu@student.lu.se`

April 19, 2016

Master's thesis work carried out at Axis Communications AB.

Supervisors: Johan Rudholm, `Johan.Rudholm@axis.com`
Magnus Mårtensson, `Magnus.Martensson@axis.com`
Jonas Skeppstedt, `Jonas.Skeppstedt@cs.lth.se`

Examiner: Per Andersson, `Per.Andersson@cs.lth.se`

Abstract

This thesis explores the communication between interconnected System-on-Chips that process images. It explores the difficulties faced when combining two heterogeneous multiprocessors to act as a single unit. It also examines the performance gain of the resulting system.

The thesis shows that not only does the resulting system decrease the bandwidth usage on the individual System-on-Chips, but it also opens the possibility of using remote processors for offloading of independent tasks.

Keywords: MSc, System-on-Chip, Embedded, Inter-connected SoC, SystemC, QEMU, TLMu

Acknowledgements

We would like to thank our supervisors at Axis, Magnus Mårtensson and Johan Rudholm, for their support and knowledge shared during our thesis work at Axis. We have also received valuable input and help from our colleagues Mikael Starvik, Rabin Vincent and Lars Viklund - for this we are very grateful.

We would also like to thank Axis and specifically Johan Ekenstierna for providing us with all the necessary material needed to complete the thesis.

We also thank our supervisor at Lund University Computer Science faculty, Jonas Skeppstedt for the invaluable input on the approach and evaluation aspects of our thesis.

Contents

1 Introduction **7**

1.0.1 Similar work . . . 7

2 Approach **9**

2.1 Division of Work . . . 10

2.2 Theory and Technologies . . . 10

2.2.1 SoC Overview . . . 10

2.2.2 Pipeline Overview . . . 10

2.2.3 Local Central Processing Units . . . 11

2.2.4 Image processing . . . 12

2.2.5 Methods of Dividing the Workload . . . 13

2.2.6 ARTPEC Driver . . . 16

2.2.7 QEMU . . . 18

2.2.8 SystemC . . . 18

2.2.9 TLMu . . . 19

2.2.10 PCI Express . . . 20

2.3 Method . . . 20

2.3.1 Overview . . . 20

2.3.2 Interconnection using QEMU . . . 21

2.3.3 Interconnection using SystemC . . . 22

2.3.4 Modifying the LCPU Setup . . . 22

2.3.5 Modifying the Slave's OS . . . 23

2.4 Implementation . . . 23

2.4.1 Introduction . . . 23

2.4.2 Image Path Through ARTPEC . . . 24

2.4.3 PCIe Bridge Driver . . . 25

2.4.4 Sending Commands Between ARTPECs . . . 27

2.4.5 SystemC PCIe Back End . . . 27

3 Evaluation **33**
3.1 Results 34
3.2 Discussion 35
3.2.1 Bandwidth 35
3.2.2 Why Choose a Dual Chip Solution? 36
3.2.3 Future Work and Improvements 36
3.2.4 Issues with QEMU 39
3.2.5 Issues with TLMu 41
3.2.6 Handling LCPU Command Results 42
3.2.7 Direction of PCIe Accesses 43
3.2.8 Alternative Implementation 44
3.2.9 Development Workflow 44
3.3 Conclusion 44

Bibliography **47**

Appendix A Glossary **53**

Appendix B ARTPEC-6 Usage **55**

Appendix C Bandwidth Calculations **57**

Chapter 1
Introduction

It is common that modern embedded systems are built on platforms with System-on-Chip (SoC) in which two or more different processors are put on a single chip to form the architecture of a heterogeneous multiprocessor.

These types of architectures provide high performance at low cost, albeit with new design challenges as well as added complexity to the software development process.

Axis's chip platform is called ARTPEC - Axis RealTime Picture Encoding Chip. It allows for communication between chips interconnected via PCIe [17]. There is currently, however, no Axis product utilizing this feature.

Each Axis camera contains one of these chips (some products contain third-party chips). The chip features hardware accelerated processing of specific tasks, such as decoding, encoding and so on. Since Axis design these chip themselves, they are cheap to manufacture.

Embedded systems applications rely more and more upon interconnected individual subsystems. As such, we believe that in the future, new systems will have functionality built using less isolated components and more by components interacting within an interconnected system.

This thesis aims to explore how best to divide the system load across two ARTPECs and show, by proof-of-concept, that it is feasible to use two interconnected chips for high-bandwidth demanding applications. In practice, this means that two of these chips will reside inside the camera, dividing up the work. The implementation is aimed at the sixth generation of Axis's image processing chip - the ARTPEC-6. Unless explicitly stated when referencing the ARTPEC, we specifically mean the ARTPEC-6.

1.0.1 Similar work

When it comes to related work, the three main articles listed in the introduction were the most similar ones we were able to find on our chosen subject. They all touch on various aspects encountered during the thesis.

- System level processor/communication co-exploration methodology for multiprocessor system-on-chip [26]

  Details the approach when designing SystemC simulations to work with both hardware and software, it was a good introduction to the mindset of writing simulated hardware. The authors also talk about the slow simulation time when using SystemC, something that we also suffered from.

- Design and implementation of an inter-chip bridge in a multi-core SoC [27]

  Contains both an explanation of an inter-chip crossbar and PCIe interface between the two chips. This was especially useful when approaching the implementation in a conceptual manner. The SystemC model that Axis uses also makes use of a crossbar - or in their case called xbar.

- Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia [23]

  The article targets parallel multimedia processing, using an object-oriented message passing model on a MultiFlex system. There are some similarities with the command interface used by Axis on their subsystems, but on a lower level. Since our implementation is on a higher level, mostly driver level, and more about the division of work - this was not as useful as we would have thought.

Chapter 2
Approach

During the first part of the thesis work, we did not have access to the actual hardware. The need for a simulated environment was obvious. Axis uses a combination of *Transaction Level Modeling* [22] (TLM) and *QEMU* [8] (an open source processor emulator) to emulate the hardware. The ability to do this is provided by TLMu - *Transaction Layer eMulator*. TLMu extends QEMU by allowing it to interact with device models built using, for example, SystemC. TLMu also makes it possible to emulate multiple CPUs in the same process [18]. This is very convenient, since QEMU is able to emulate standard hardware platforms such as Intel x86, ARM, CRIS et cetera. As such, it is possible to emulate standard processors and components using QEMU and build the non-standard components using SystemC.

Like other companies [25], Axis has found that emulating the hardware shortens the lead time for complex embedded projects.

We have examined the ARTPEC image processing pipeline to decide how to partition work between the chips, or how to *split* the pipeline. No matter which split we would have chosen, we decided early on to have a master-slave relationship between the two chips, where one chip controls the other. We will show that a vertical split of the pipeline is both the easiest and most rewarding approach in terms of time invested in the implementation.

Using the Axis implementation of TLMu, we implemented a simplified *Peripheral Component Interconnect Express* (PCIe) interconnection between the two simulation instances. The interconnection went through a series of different implementations before we found the correct approach.

We tried implementing the bridge both within QEMU and SystemC. The approach is, however, similar as both implementations use a memory mapped region and send data between the simulation instances via POSIX sockets as will be shown in this chapter.

The ARTPEC Linux driver, which acts as a coordinator between the hardware and image capture requests, was modified to account for the interconnection. The slave was in turn stripped to only initiate the LCPUs (mentioned in section 2.2.3) and PCIe.

2.1 Division of Work

The work throghout the thesis has been equally divided between the both of us. With this, we mean that every design choice was thoroughly discussed together, before any implementation. During some parts of the implementation, mainly during debugging, it was hard to parallelize the work. We concluded that during these periods, it was best that Andrée focused on debugging code while Felix focused on the theoretical aspects of the thesis. These aspects include, to name a few: calculation of bandwidth usage and other metrics needed to evaluate the implementation.

2.2 Theory and Technologies

2.2.1 SoC Overview

The ARTPEC-6 SoC contains a dual core 1 GHz ARM Cortex-A9 MPCore CPU, henceforth called the MCPU (Main Central Processing Unit). The MCPU is used for generic calculations and controlling the different subsystems on the SoC. Most - but not all - subsystems contain one or more processors called Local Central Processing Units. These LCPUs are used to perform specific tasks specified by the MCPU, see section 2.2.3. There are a total of nine different subsystems, each in charge of different commands, such as cryptography, graphics processing, peripherals et cetera. The SoC also contains electrical interfaces like ethernet and PCIe. An overview of the SoC is shown in figure 2.1.

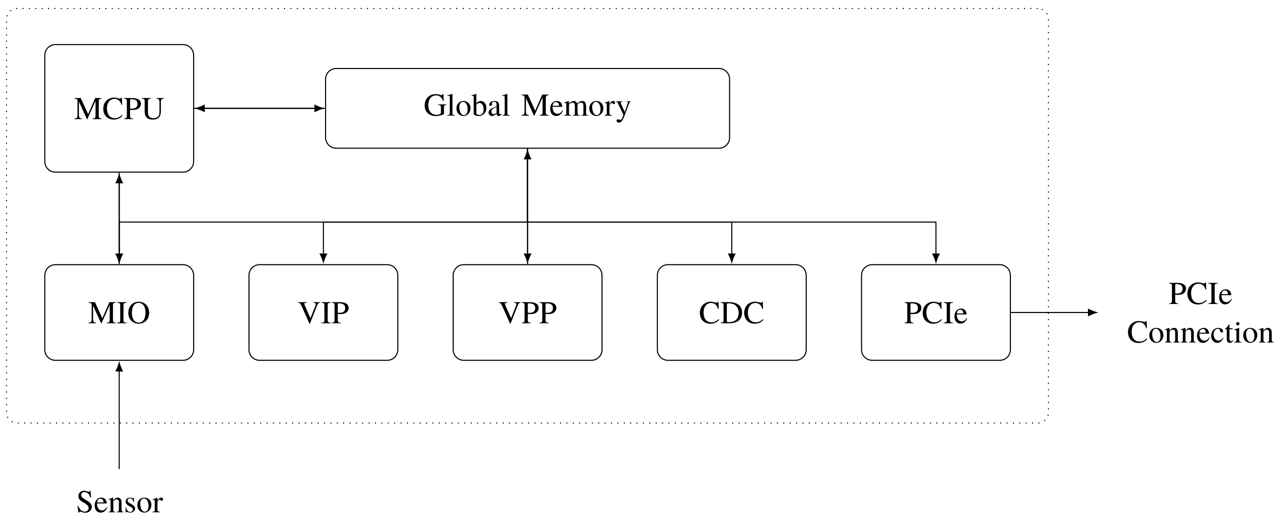


Figure 2.1: Overview of SoC

2.2.2 Pipeline Overview

The MCPU offloads different tasks by sending commands to the different subsystem queues. The different subsystems will then process these commands individually. Four of the available subsystems are always used when processing an image throughout the pipeline as shown in figure 2.2.

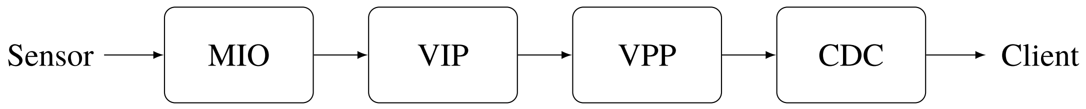


Figure 2.2: Image processing subsystems on the ARTPEC and a receiving client

- **MIO**: Media Input/Output - handles capturing images and media from the different sensors
- **VIP**: contains the Image Processing Pipeline (IPP). The VIP performs various tasks like applying filters, performing noise reduction and the like. The order of filters and processing in the IPP is proprietary and confidential
- **VPP**: Video Post Processing - this unit performs transformations on the finished image; id est transformations like scaling and cropping
- **CDC**: Compression and decompression - handles encoding and decoding between different encoding and compression schemes
- **Client**: A client that receives the encoded frames from the CDC, such as a web view or a video player

The other subsystems (managing peripherals and cryptography for instance) are used by the MCPU when needed, and do not typically require a lot of bandwidth or performance compared to the subsystems involved in the image processing. For this reason they will not be discussed in detail throughout this thesis.

2.2.3 Local Central Processing Units

The ARTPEC contains multiple *Local Central Processing Units* (LCPUs). These units are part of a subsystem that performs specific tasks orchestrated by the MCPU.

Each LCPU consists of a CRIS processor with its own local RAM and possibly standard peripheral units, exempli gratia FPUs et cetera. CRIS stands for “Code Reduced Instruction Set” and is the CPU architecture developed by Axis [1].

The MCPU communicates with the LCPUs via a well-defined API. This API uses command and response queues to communicate with the LCPUs. The MCPU can queue several tasks in an LCPU, so that the latter may perform them back-to-back.

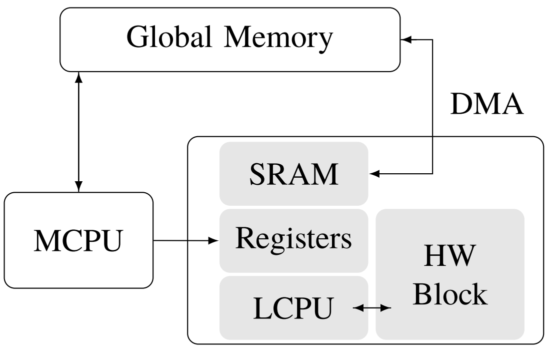


Figure 2.3: A subsystem communicating with the MCPU

In figure 2.3, the MCPU is communicating with an LCPU subsystem. The arrows indicate the direction of data between the different parts. As can be seen, both the MCPU and the LCPU can place data into the global memory. The LCPU, however, needs to use DMA to transfer data between the global memory and its SRAM - this if further explained in section 2.2.4. As previously stated, the MCPU can orchestrate the tasks performed by the LCPU. It does this by writing and reading specific registers in the LCPU. In turn, the LCPU controls a hardware accelerated block to perform heavy lifting.

2.2.4 Image processing

The MIO subsystem handles the image capture and transfers the data to the VIP where it is processed and enhanced in various ways and then transferred to the VPP subsystem. In the VPP subsystem the image is (if needed) scaled and transformed, which is the last processing step of the image. Finally the image is sent to the CDC subsystem where the image is encoded to another format such as H264 [6].

Command Queues

The LCPU has access to two First-in-First-Out (FIFO) queues. A command queue, and a response queue. The MCPU can insert commands in the first queue and send an Interrupt Request (IRQ) to the LCPU, informing it of inserted commands. The commands are described as C-structs in global memory. A pointer to the struct is passed to the commands queue of an LCPU.

When an LCPU has completed a command, it will insert a response in the response FIFO queue. The MCPU can be informed of the existence of any responses by triggering a response IRQ. The response contains a pointer to the original C-struct detailing the command.

The MCPU must *only* communicate with the LCPUs using a well-defined API, working as described above.

To keep the subsystem busy, many commands should be issued before expecting a response from the first. The LCPU may execute the commands out of order and in parallel as there is support for multiple parallel software pipelines.

Direct Memory Access

Direct Memory Access (DMA) allows the LCPUs to access resources outside of the subsystem, e.g. global memory, independent of the MCPU [20, Chapter 15]. The DMA on the ARTPEC between LCPUs and global memory, is not optimized for high-performance, since transfers are assumed to contain limited amounts of data at low transfer rates.

All addressing is aligned to a 4 byte boundary by ignoring the least two significant bits. For efficiency, external addressing targeting PCIe or global memory should be aligned to a 32 byte boundary. Transfer lengths should also be a multiple of 32 bytes, since the transfer will consume a multiple of 32 bytes. If we e.g. choose to transfer 33 bytes, 64 will be transferred.

IRQs are used to inform the involved parties when DMA is ready for a new transfer.

Direct Data Transfer Between Subsystems

The ARTPEC chip has the ability to send data directly between certain subsystems. This is achieved by using a FIFO connection between these subsystems. This concept is known internally at Axis as *flow*. There is flow between the MIO and the VIP as well as between the VIP and the VPP.

When flow is used, the data does not need to enter global memory to propagate through the pipeline until it reaches the CDC. This results in lower bandwidth usage to and from the global memory.

The other way of transferring data between the subsystems is to bounce the data to global memory, into different bounce buffers. The MCPU allocates the buffers needed by each subsystem; capture, process and scale buffers to name a few. The subsystems are then able to access these buffers using DMA.

Flow has a couple of limitations - when there are multiple sensors, flow cannot be utilized. This is because of flow's nature, it is not implemented in a way that supports more than one flow of data from the sensor outputs. Another issue is that if the subsystems cannot keep up with the rate of captured images - flow will be disabled and the system will fall back to using bounce buffers.

2.2.5 Methods of Dividing the Workload

When evaluating how best to split the pipeline as seen in figure 2.2, we've considered three options - splitting the pipeline either horizontally, vertically or by every other frame. Each of these options create fundamentally different problems.

Horizontal Split

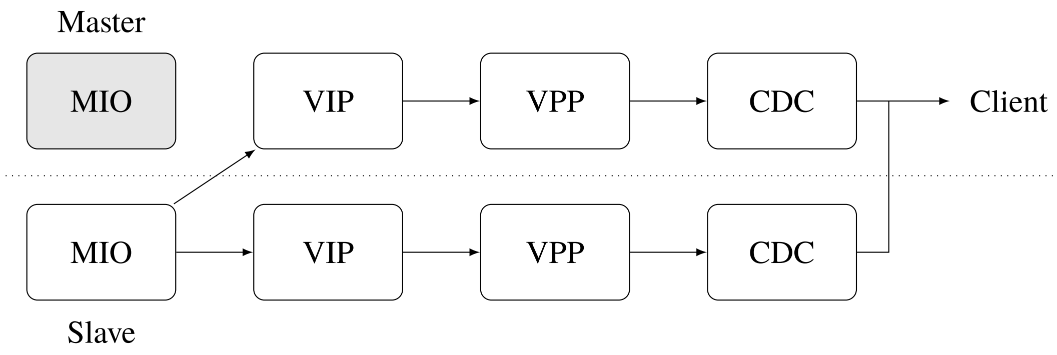


Figure 2.4: Horizontal split, where the Master's MIO remains unused.

Splitting the pipeline horizontally would mean dividing the images into two parts, and then letting the two ARTPECs process each part individually - merging them at the CDC subsystems, as seen in figure 2.4. This would mean that both ARTPECs would share the load nearly equally. There would, however, be timing issues regarding the final merging process. We would have to ensure that the correct images are stitched together. We would also need to be certain that the VIP subsystem is not performing image processing in a way that would distort the edges of the separate parts of the image. Unfortunately, the VIP is proprietary and confidential. Modifying said code to ensure that images aren't distorted is outside the scope of this thesis.

Even if we assume that the VIP does not interfere with the images - selling features like object recognition would be seriously affected by splitting the image. This might be solvable by letting the two systems process partially overlapping images and blending them together in the end. Another master's thesis project at Axis, created a 360 degree panorama for the ARTPEC-5. Their findings indicate that it would be possible to do this in real time for lower resolutions [24]. As the goal of using two ARTPEC-6s is to be able to handle high-bandwidth demanding applications such as 4K-resolution videos, this seems infeasible or at best hard to estimate the resulting framerate.

Vertical Split

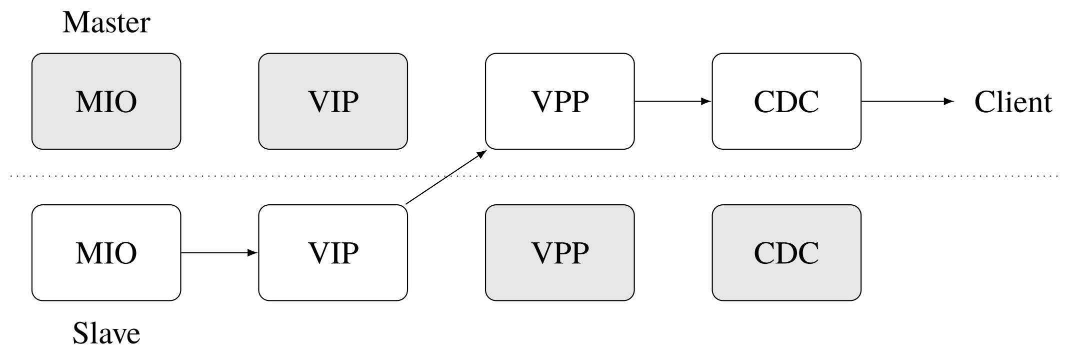


Figure 2.5: Vertical split, where the Master's MIO and VIP, as well as the Slave's VPP and CDC remain unused.

Splitting the platform vertically would mean to divide the work performed by different subsystems to different platforms. Looking at the pipeline in figure 2.2. It might seem as though it would be natural to split the pipeline down the middle after the VIP is done with the image processing, as illustrated in figure 2.5.

First off, the different subsystems might not take equally long time to perform their respective tasks or use the same amount of resources to perform these. Also as the image does not need to enter global memory until the encoding process; it would hypothetically be beneficial to simply move the image to the other system's memory directly and let it perform the encoding and delivery.

Using flow, a possible solution would be to perform the image capture (MIO) and the image processing (VIP) on the slave ARTPEC. Then transfer the image via PCIe to the VPP on the master ARTPEC. Thus the master ARTPEC would receive the final image to be scaled and encoded.

However, as mentioned in section 2.2.4, flow cannot be used in all scenarios, the data might be forced to enter global memory anyway. This simply means that the path of the data is extended, but the overall idea is the same.

Every Other Frame

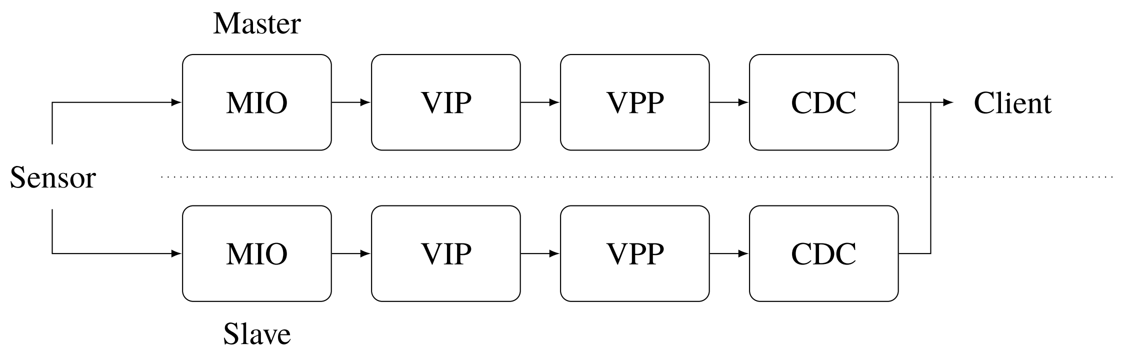


Figure 2.6: A split where even frames are sent to Master, and odd frames are sent to Slave.

Assuming that the different ARTPECs share a sensor and that they can then get every other frame at an early stage in the pipeline (i.e. before the VIP) - the driver would need little modification. The *master* ARTPEC would encode even frames and the *slave* would encode odd frames. An illustration of where the sensor output is split between the two MIO subsystems can seen in figure 2.6.

The problem with this approach arises when encoding the stream. Compression methods like H264 use a base image frame (I-frame) in the beginning of the stream - and can then use delta frames (P-frames) to encode the subsequent images.

The encoding subsystem (CDC) would also need to be tweaked to allow two ARTPECs to coordinate the use of I- and P-frames in the encoding process. It would be possible to only send the I-frames in a proof-of-concept, but the loss in compression would be complete.

Chosen Method

The complexity of performing a horizontal split and the loss of compression in the every other frame proof-of-concept drove us towards favoring a vertical split. The vertical split, albeit not equally load balancing, should be able to increase the work capacity of the ARTPEC platform.

2.2.6 ARTPEC Driver

The ARTPEC driver is a GNU/Linux kernel module developed by Axis. It basically consists of two kernel modules; the ARTPEC-6 UDL (User Defined Logic) and the ARTPEC-6 module. The UDL module takes care of probing, loading and interfacing with the LCPUs, in other words, it is the module closest to the LCPUs. The UDL module also defines an interface for interacting with LCPUs, an interface used by other modules such as the ARTPEC-6 module. The ARTPEC-6 module defines the interface used by user space programs. The driver uses `ioctl` [20] to allow interaction with the hardware from user-space. Examples of `ioctl` calls are adding/removing encode requests, allocating video memory buffers, getting/setting camera settings. A simple example showing the basic usage of the ARTPEC-6 module is available in appendix B.

Design

The driver uses an object-oriented approach to simplify the addition of new functionality. The process of generating an image consists of more or less four steps:

1. Capturing the image, performed by the MIO subsystem in hardware
2. Processing the image using the VIP
3. Scaling streams into desired sizes, performed by the VPP subsystem in hardware
4. Encoding the streams into the desired format, performed by the CDC subsystem in hardware

The driver follows the same steps, with some added complexity and behavior, as illustrated by figure 2.2. Because of the use of third party encoding blocks, the image must enter primary memory between steps 3 and 4. If the ARTPEC only has one sensor attached, however, steps 1 through 3 may be performed in parallel, using the FIFO mode described in section 2.2.5.

Capturing an Image

There exist a number of ways to capture an image in the driver. The driver can interact with several different input devices, referred to as `input_modules` in the code. When an ARTPEC has multiple image sensors attached, there is one `input_module` for each image sensor.

The driver allocates multiple capture buffers for each source to be able to capture data from the sensor and process it simultaneously. Each image contains 1.5 – 2 bytes per pixel and is stored in video memory buffer. Space for these buffers needs to be reserved at boot-time.

Once an image is captured and placed in a video memory buffer, the VIP is notified and starts processing the buffer. As to not overwrite the buffer with the next image, multiple buffers per input module are needed.

Processing

The processing step is offloaded to the VIP subsystem and handled by its LCPU. This step processes the image by adding filters like noise reduction, tone mapping, white balance et cetera. When applying these filters, the image may be repeatedly transferred between global memory and the VIP's hardware block and may, therefore, utilize a high amount of bandwidth.

Scaling

The VPP performs actions like scaling, rotating, adding overlays, and performing aspect correction before passing the data on to the encoder. For efficiency, the VPP will identify jobs that require the same properties, such as resolution, and only perform these actions once.

It is worth noting that some actions are expensive, like 90° rotations. These are considered expensive, since the VPP needs to perform two passes on the data. This means that as the VPP works on the second pass, it cannot receive any data from the VIP; which means that flow cannot be enabled and that the bandwidth usage from global memory is increased.

Encoding

There are several encoders on the ARTPEC chip e.g. JPEG and H.264. These are implemented in hardware and can put the result in a non-contiguous memory buffer, meaning that this memory can be allocated by user-space.

The encoders have their own scheduler which tries to find jobs that can be performed by the encoders immediately.

2.2.7 QEMU

QEMU, which stands for Quick Emulator, is an open-source hosted virtual machine monitor. Through binary translation, it can simulate device models such as CPUs and various standard hardware components e.g. network cards. When used with Kernel Virtual Machine (KVM) [7], it can achieve near native speed for virtual machines [8].

QEMU allows the developer to create his or her own hardware models. It is possible to use standard Linux drivers and emulate the device via the models created by the developer. Many CPU vendors have contributed models emulating different different processor architectures. There is x86 support as well as ARM support to name a few. If the QEMU instance runs the same architecture as the host system, it can execute the instructions on the host CPU without translation [8].

At Axis, QEMU is used to write hardware models for the more standard components such as network cards and other peripherals.

2.2.8 SystemC

SystemC is a C++ framework which allows for hardware to be simulated in an event-driven fashion [19]. In contrast to QEMU, the basis of SystemC is events. SystemC can be used to emulate the different timings and latencies of the expected hardware. SystemC allows different *modules* to be run concurrently and provides different mechanisms to communicate with surrounding components. These mechanisms have real world equivalents e.g. SystemC's signals correspond to wires.

Axis uses the fact that SystemC provides support for *Transaction Level Modeling* (TLM) [16] to simulate upcoming ARTPECs. TLM abstracts the communication between models away from their implementation, this allows the system designer to experiment with different bus architectures without having to recode models that interact with said buses. The communication between the models is provided by SystemC channels.

Axis has chosen to emulate their platforms in SystemC, which allows the implementation of the CPU peripherals and buses with their expected timings and latencies. The platform contains several modules to emulate the subsystems of the ARTPEC (figure 2.2).

2.2.9 TLMu

As aforementioned, Axis uses QEMU to simulate standard components. QEMU is used in conjunction with SystemC and the unison between these frameworks is called *Transaction Level eMulator* (TLMu) [18]. TLMu establishes a way to communicate between SystemC components and QEMU. While there are similarities between QEMU and SystemC, the modeling goals are different. QEMU focuses on simulating a system with a single CPU, while SystemC is much more hardware oriented - supporting things like latencies and timings.

The SystemC modules are interconnected with the platform using ports. Each subsystem also contains a crossbar with the memory map for the subsystem as well as a connection to one or several QEMU instances containing the subsystem's LCPUs as shown in figure 2.7.

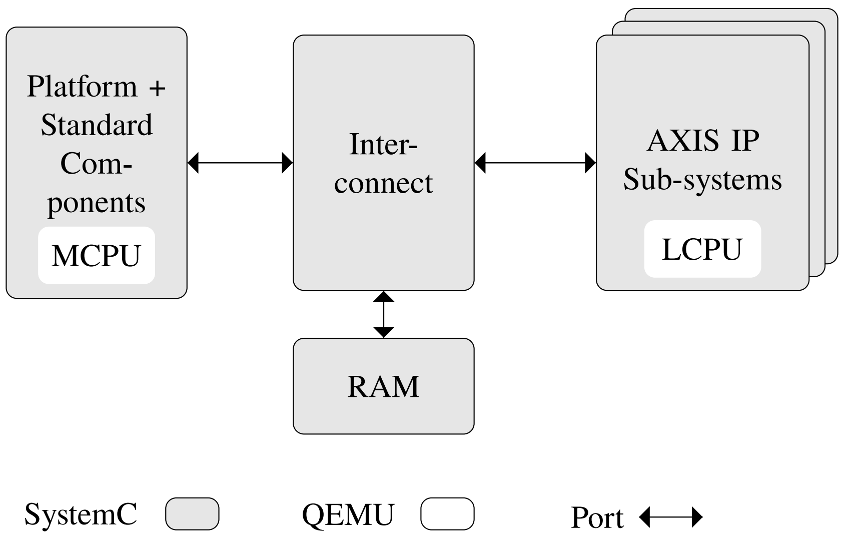


Figure 2.7: Schematic of SystemC interaction with QEMU alias TLMu

All the QEMU CPU cores interface with SystemC through a SystemC module that exposes a set of sockets. To run multiple instances of the TLMu system on a single host, these ports need to be configurable as the same port can't be bound multiple times.

2.2.10 PCI Express

PCI Express (PCIe) is an expansion bus, based on the previous PCI bus standard, and is used in order to attach devices and other peripherals in a computer [17]. The system will consist of one root complex (RC) which will act as the host, and multiple end points (EP) which are clients. On the ARTPEC, two ARTPEC boards can be connected directly using PCIe, but as long as the PCIe driver is properly implemented, this bus could be used in order to connect other devices. Using a PCIe switch enables multiple devices to be connected to a single PCIe port.

In order to access the other PCIe device, the RC can either map the device into the I/O port address space or the memory mapped address space, the latter is used in the dual ARTPEC solution. The RC and EP are able to map different regions of the PCIe addresses to the other device's address space. This enables the RC to communicate and control the EP, a concept which fits nicely into the master-slave configuration intended for the dual ARTPEC solution.

An important aspect of PCIe is the fact that reads are slower than writes [13]. In order to read, the reader must send a request containing no payload (meaning that there is only overhead), which in turn will cause the receiver to respond with the data. Writes however, only require one write request; this is important to consider when interfacing with PCIe devices.

Another feature required is the ability to forward and emit interrupts over the PCIe bus, enabling the RC to seamlessly access devices which require interrupts, such as the LCPUs of the ARTPEC. This can either be implemented using *legacy interrupts* or Message Signaled Interrupts (MSI) [15]. Legacy interrupts are simpler and are routed to the RC's interrupt controller, requiring each type of interrupt to use a specific interrupt number (which are a limited resource [20]). MSI however, is more versatile protocol and might be a better alternative.

Message Signaled Interrupts (MSI) enables the hardware to support up to 32 interrupts, while only requiring one physical interrupt lane. The MSI message contains information about which interrupt number triggered the message to be sent, enabling the interrupt handler to discern what to do.

2.3 Method

2.3.1 Overview

As aforementioned, the proof-of-concept was implemented using simulated hardware. As a consequence, the approach is a bit different than if we had implemented it with direct access to the hardware. Nevertheless, our approach can be summarized in the following steps:

- **Decide were to split pipeline:** we decided upon splitting the pipeline in a vertical split as mentioned in section 2.2.5. The details of this choice are explained in the same section. This split effectively turns one ARTPEC into a *master* and the other into a *slave*.

- **Simulate the PCIe connection:** the actual hardware will be linked via PCIe, since we are using a simulated environment we need to implement something that acts like the PCIe bridge would. That is to say, we do not need to implement it *exactly* like PCIe, we can take some shortcuts.

- **Modify the ARTPEC driver:** the master's ARTPEC driver is the main orchestrator of what happens in the pipeline. As such, it needs to be modified in order to properly use the remote LCPUs.

- **Minimize the slave's OS:** we remove all unnecessary modules to make the slave as small a system as possible. Things that are left on the slave should only be things that do not interfere with the master driver and if possible should alleviate the complexity of our solution.

With the actual hardware, the simulated PCIe connection can be dropped. Axis has already implemented drivers for the master and slave PCIe buses, as such the main problems to solve there are the first and last step in the list above.

2.3.2 Interconnection using QEMU

During our thesis work, we never had access to actual hardware, as Axis were still testing and developing the drivers and firmware. We needed to simulate the interconnection to perform the pipeline split. The ARTPEC-6 can be connected via PCIe as shown in figure 2.8. Where the idea is that one of these acts as a *master*, issuing commands to the *slave* ARTPEC. Only one of these chips would need to run an entire Linux system. Using QEMU we emulated a PCIe driver and used a socket to act as the bridge (as in figure 2.9).

Later this implementation was abandoned in favor of a SystemC implementation as explained in section 2.3.3.

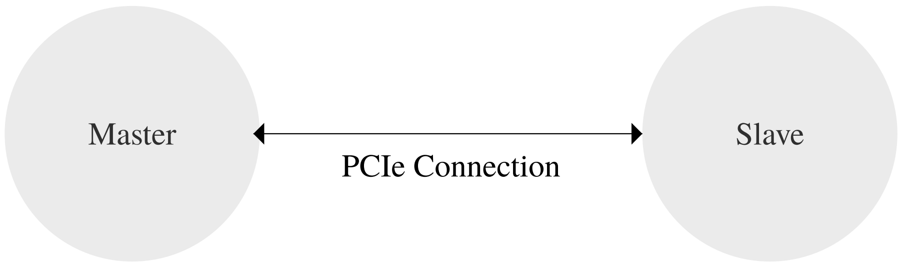


Figure 2.8: ARTPECs interconnected via PCIe

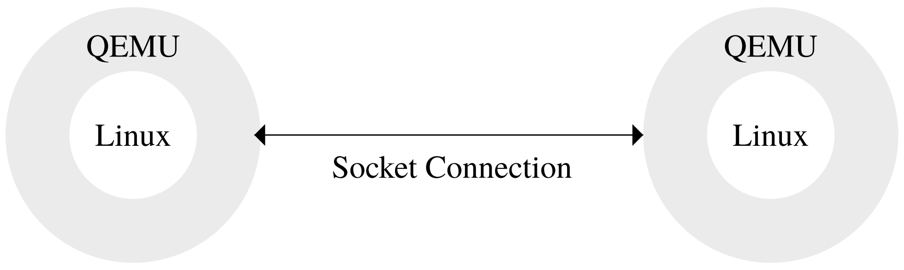


Figure 2.9: Interconnected ARTPECs simulated with TLMu

2.3.3 Interconnection using SystemC

Later on during our thesis work, we realized that the interconnection was not possible to implement using QEMU. Sending data using QEMU is fine, however, a thread not created by SystemC is not allowed to use the various functions that simulate hardware operations, such as the memory read and writes required by the PCIe simulation. The reason being that the memory read functions are simulated using delays (in this case the `wait` function) and these functions can only operate properly when running in a SystemC thread. This forced us to instead implement the PCIe module purely in SystemC and making sure that the receiving socket thread was properly created using the SystemC framework.

2.3.4 Modifying the LCPU Setup

The ARTPEC driver on the master needed to be modified in order to issue commands to the slave. The driver contains several levels of abstractions, from user space the user may simply issue `ioctl` calls to control the image capture. As such, our first stab at adapting the driver was to somehow wrap these calls and send them to the slave ARTPEC. This method, however, would need two running Linux systems.

The Linux device tree has support for changing the drivers associated with the different LCPUs as well as changing their physical start and end addresses.

Axis supplies remote versions of their LCPU drivers, that support probing using PCIe. Specifying the use of these can be achieved by simply changing a constant. We, however, realized that it would be easier for us to let the driver think that it was communicating with a local CPU and just change the target addresses to addresses we had mapped to PCIe in memory. This means that our PCIe simulation could be kept simple and not compatible with the generic PCIe driver [11] that the ARTPEC-6 module relies on.

As such, we changed the addresses of the MIO and the VIP. These LCPUs were now mapped to an area in memory that we had performed `ioremap` on. `ioremap` is a function in the Linux kernel which returns a virtual address that can be used to access I/O memory regions [20].

2.3.5 Modifying the Slave's OS

The slave does not really need to run a full OS kernel. It is, however, preferential to let the regular system load the firmware for the LCPUs and perform setup of the different systems needed on the slave chip. As this allows us to focus more of our attention to how we handle splitting the pipeline - we decided to let the slave keep its OS for the time being.

In order to focus on the pipeline split, we decided to keep a modified version of the ARTPEC-6 UDL module. It still flashes the LCPUs, but does not setup debugging and other operations that can interact with the LCPU after the setup has been completed, i.e. the UDL does something similar to what a primitive OS would do on a real chip. The ARTPEC-6 module, however, is completely removed, since no program should be interfacing with the LCPUs except the master system.

Additionally, the LCPUs firmware had to be modified in order to support MSI. The reason for using MSI instead of legacy interrupts is simply that the ARTPEC-6 driver was already prepared with partial (but untested) support for MSI.

2.4 Implementation

2.4.1 Introduction

The final implementation will make the captured image travel a very different way through the pipeline than we originally imagined, or at least by different means. The road to this implementation is detailed throughout this chapter, but we will start by explaining the resulting implementation in a high-level manner.

2.4.2 Image Path Through ARTPEC

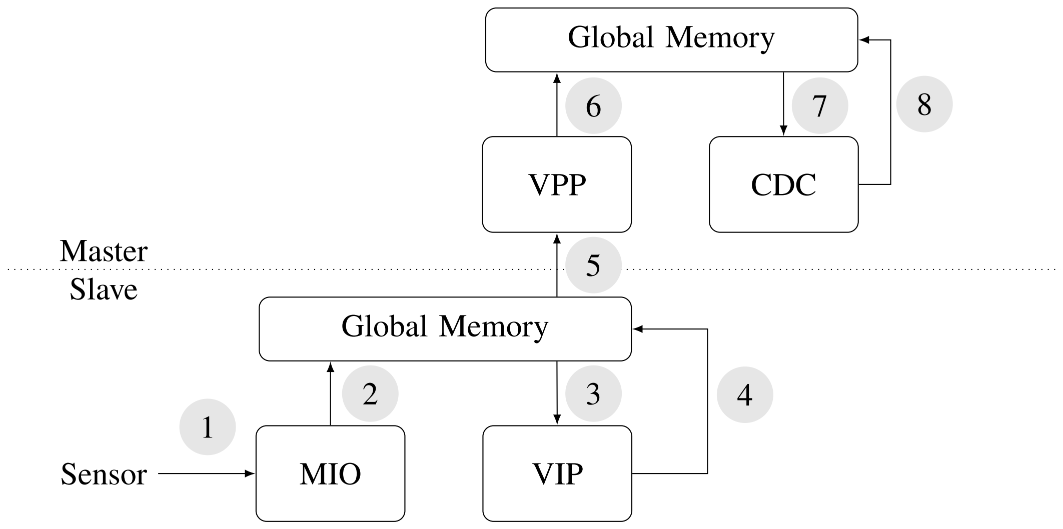


Figure 2.10: Image path through the ARTPEC-6 Dual Chip Solution without flow

1. Using a Python script we are able to inject previously captured images into the MIO from a source on the host. The script opens a socket connection to the SystemC model, which then injects the image directly into the MIO subsystem. It is also possible to use a real camera and then bounce the images via the host to the simulation using the same script; this enables the simulation to use a live feed.

2. The MIO will place the captured image in a *capture buffer* in the slave's global memory via DMA. It will then notify the ARTPEC driver that a captured image is available.

3. The ARTPEC driver will then inform the VIP of the image to be processed. The VIP will process the image contained in the capture buffer.

4. The VIP will then place the resulting chunk of the image back in the master's global memory, but in a different buffer known as a *process buffer*. Once it is, done with all chunks it will inform the driver.

5. The VPP will need access to the image from the slave ARTPEC's global memory. As such, chunks of the image will be transferred to the VPP, where the VPP can work on the image. This behavior is the same as what occurs in step *3*.

6. The VPP will place the resulting chunk in a *scale buffer* which resides on the master. There might also be an extra buffer allocated, if the scaler is told to rotate the image. No matter the configuration, once the VPP is done with all chunks - the driver is notified.

7. There are several different coding blocks, H264, JPEG et cetera. The image from the scale buffer is processed and placed within one of the different coding buffers. After this step, the image is ready for consumption by the user space application.

8. The final image will be placed back in global memory and made available to the client application.

The choices that have led to this implementation are shown throughout this chapter. The path in figure 2.10 differs if flow is used, paths *2* and *3* are merged if flow between MIO and VIP is used, which means that the slave global memory is bypassed. Similarly paths *4* and *5* are merged if flow between VIP and VPP is used.

2.4.3 PCIe Bridge Driver

As mentioned in section 2.3.1, the emulated version of the driver is supposed to function like PCIe. For our proof-of-concept implementation, however, the driver was implemented to be as simple as possible. The bridge driver uses the ARTPEC driver to map memory and a backend provided by the simulation. The backend handles the aspects of the driver concerned with emulating PCIe. Our original approach intended to use QEMU as the backend - unfortunately, this proved infeasible. The original implementation using QEMU served as an inspiration to how the final driver was implemented using SystemC. As such, the details of this implementation, as well as the final implementation, are described below.

ARTPEC Driver

Via `ioremap`, the ARTPEC driver, on the master, maps a portion of the physical memory to virtual memory addressable by the kernel [20, Chapter 9]. The driver also registers an interrupt handler for use when data has been made available on the bus.

When an interrupt is raised, the driver can read the data and clear the interrupt if applicable.

The memory mapped region makes it possible to access different parts of the remote system. Thankfully, the ARTPEC driver does not need to implement PCIe emulation, since this can be handled in the backend.

QEMU Back End

The QEMU back end acts as the link between two instances of the emulator. These instances can be run on different machines and as such, we chose the means of intercommunication to be POSIX sockets. This will of course not be needed on the actual hardware.

The QEMU backend registers itself as the hardware for the portion of the physical memory that was mapped by `ioremap` in the driver. Whenever the driver wants to read from or write to said portion of the memory - two functions are invoked in the QEMU backend listed in figure 2.11.

```
/**
 * Function invoked when the driver wants to read from the mapped
 * physical memory
 *
 * @param state - the state of the model
 * @param offset - the offset invoked by a read i.e. 'ptr[offset]'
 *
 * @return the value read from "memory"
 */
uint32_t iomodule_read(void *state, target_phys_addr_t offset);

/**
 * Function invoked when the driver wants to write to the mapped
 * physical memory
 *
 * @param state  - the state of the model
 * @param offset - the offset invoked by a read i.e. 'ptr[offset]'
 * @param val    - the value to be written
 */
void iomodule_write(void *state, target_phys_addr_t offset, uint32_t val);
```

Figure 2.11: Driver Back End functions in QEMU environment

When `iomodule_write` is invoked, the offset is checked to see whether it was a command or a data write. On a data write, the written data is stored in the module's state and subsequentially written to the remote chip.

The UNIX socket is connected to the other instance of the emulator. When the receiving instance has data available, it writes the data into its model and raises an IRQ. This means that the receiving side will be notified via IRQ, when data has been written to its memory.

As such, the receiving ARTPEC can be made aware that there is data or a command available and act accordingly.

To make sure that the receiving socket reads the appropriate amount of bytes, a struct defined below in figure 2.12.

```
typedef struct {
        size_t   size;
        uint32_t data[];
} iomodule_header;
```

Figure 2.12: Header with data sent over socket between QEMU Back Ends

When the receiving instance starts reading, it can read the size variable and determine how much it should read before alerting the driver of the data by raising the IRQ. Doing it this way ensures that the driver has access to all of the data being sent by the remote ARTPEC.

2.4.4 Sending Commands Between ARTPECs

Our first idea was to wrap the `ioctl` calls and send them directly to the slave ARTPEC. The drawback with doing this, however, is that the ARTPEC driver will perform several LCPU commands on one `ioctl` call. As such, we decided that it would be easier to simply wrap all commands sent to certain LCPUs and send them to the remote ARTPEC. Please note that commands sent to the LCPUs are in fact register writes made to the LCPU command FIFO-queue. The actual commands will then be fetched by the LCPU via a DMA transfer as mentioned in section 2.2.4.

Wrapping LCPU Commands

As aforementioned, commands are sent to the LCPUs from the MCPU. Since the LCPUs cannot address main memory directly, they need to perform DMA to get the command struct sent by the MCPU. This poses a problem to us, since the LCPUs in the MIO and VIP need to be unaware of the fact that the commands have been sent from a remote MCPU. Thus, when fetching the command structs via DMA, the LCPU needs to work as if the memory it is fetching is on the same chip.

Therefore, the idea is to send the relevant commands via the socket in QEMU to the slave MCPU. The slave is then to re-create the commands and execute them on its MIO or VIP. Once the command is done, the slave would need to send the resulting data back to the master MCPU and place it in memory at an address expected by the ARTPEC driver. Once this is done, we would need to trigger the appropriate callback in the ARTPEC driver by sending an interrupt request.

As can be seen from this approach, we would need to wrap a lot of different driver functions - and in turn, the driver would need to be very different on the separate systems.

Direct addressing over PCIe

After the above attempts, we decided that it might be best to simply address the LCPUs over PCIe directly. Doing this results in us not having to wrap anything but redirecting the actual register reads and writes to the LCPUs.

2.4.5 SystemC PCIe Back End

The SystemC model is more complicated than the original idea we had, which solely involved QEMU, but the general idea is the same. Just like in the case of QEMU we need to capture all reads and writes to the specific PCIe addresses. The way Axis has set up their SystemC models is that all models and interconnects (basically a model consisting of other models) are connected using so-called *xbars* (also known as crossbars, a term used by similar projects [28]). Xbars are used to wire different blocks together, where each model can retrieve a *target* or *initiator* socket connected to the model by specifying the name of the target model. The target socket is used to read and write data to the other model, while the initiator socket is used to handle reads and writes from the other model.

- Every interconnect defines an `ic_to_pf` (interconnect-to-platform) xbar, it handles all outgoing requests from the interconnect to the platform. This is mapped to the whole available address space 0x0–4GB. Each internal model is registered to this xbar by name, and binds their initiator socket (called `<model>_mem`) to the xbar. The interconnect then exports the initiator socket `mem` which is bound to the target socket `pf` from `ic_to_pf`. This means that other models will be able to make requests to this interconnect by using the publicly available `mem` socket, which itself is connected to other xbars from internal models.

- Every interconnect also defines an `ic_from_pf` xbar, it handles all incoming requests from the platform to the interconnect. The interconnect binds each internal model target socket (called `<model>_ctrl`) to the `ic_from_pf` xbar by model name. It also defines the address mappings corresponding to each model's target socket. Finally the target socket `ctrl` which is bound to the initiator socket `pf` from `ic_from_pf` is exported. In order for this interconnect to be able to make requests to other models, they have to bind the target socket `ctrl` from this model.

This effectively separates all interconnects and models from each other, making the overall system very modular.

At first, the PCIe module was integrated with the interconnect like the other CRIS subsystems. This, however, makes efficient DMA impossible from the CRIS to PCIe because of the way DMA is implemented in the simulator. In the simulator, the Direct Memory Interface (DMI) is used to access memory from the CRIS subsystems. DMI tries to use DMA when possible, otherwise it falls back to using the QEMU memory access system, resulting in 4-byte transfers. This is very slow compared to DMA. As seen in figure 2.13, there is no direct memory path available from the interconnect to the PCIe. This is because PCIe resides in the interconnect and systems inside of it are not directly connected - they are merely grouped. Any DMI performed will go through the `ic_to_pf`, to the `platform` (QEMU), to the `to_from_pf` and back to PCIe.

In order to make DMA possible, another xbar was added as seen in figure 2.14. By adding paths from `ic_to_pf` and `to_from_pf` directly to PCIe, we are able to makes sure that the subsystems can take a shortcut to the PCIe's memory without involving the platform.

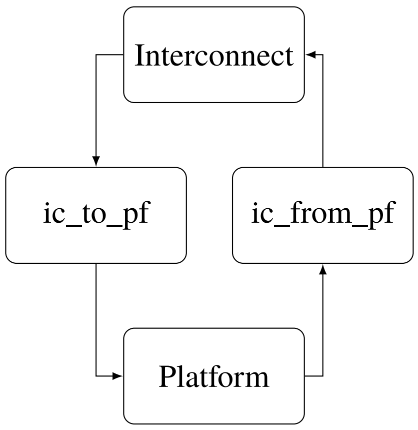


Figure 2.13: xbar connections, where PCIe resides in the interconnect.

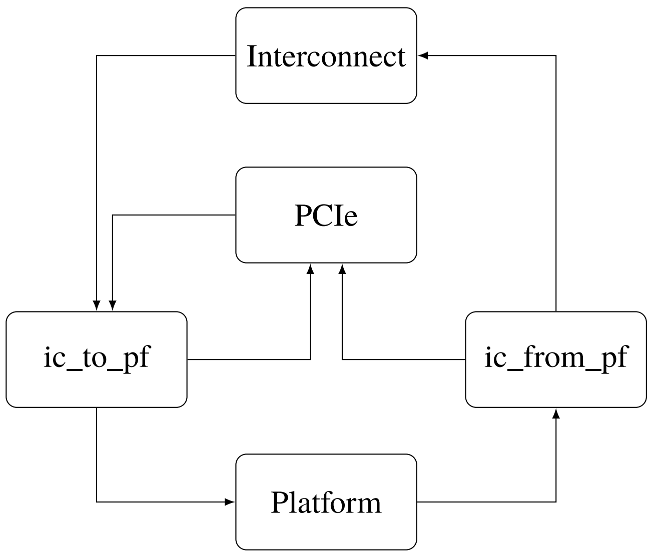


Figure 2.14: xbar connections, where PCIe is separated from the interconnect.

Memory Map of Remote LCPUs

We created a subsystem in SystemC to catch writes to remote LCPUs. IO accesses are caught by being targeted at a predefined range of physical memory. Looking at the source for driver setup in the Axis Linux repository, we found that the VIP precedes the MIO in memory. Upon this discovery, we concluded that it would be easier to mirror the original memory map in our mapped area. Each subsystem has `32kB` allocated for register reads and writes as can be seen in figure 2.15.

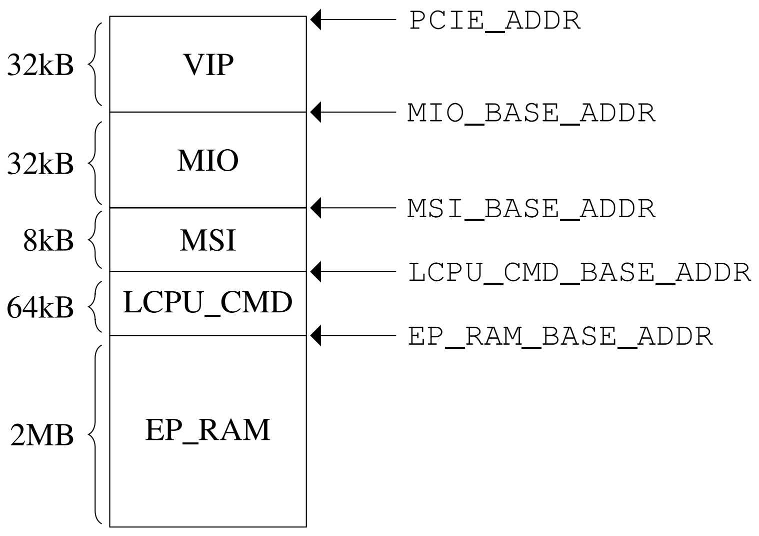


Figure 2.15: Root Complex's memory mapping

VIP this area contains the register mappings for the VIP subsystem

MIO this area contains the register mappings for the MIO subsystem

MSI this area contains the register mappings needed to pass MSI messages from slave to master

LCPU_CMD this area is used to allocate LCPU command structs that can be addressed by both the slave and the master

EP_RAM this area is mapped to a portion of the slave's global memory. It is addressable from the master ARTPEC

Communicating with target SystemC Process

Both the root complex and the endpoint have a thread waiting for input from their counterpart. The endpoint, alias slave, acts as the server, waiting for a connection from the root complex. Once the connection has been established, however, they perform the same actions - i.e. sending output and waiting for input on the socket.

The simplified PCIe connection will wait for reads and writes and propagate them across the socket if applicable.

Setting Up Remote LCPUs

The MIO and VIP subsystems on the slave were compiled with MSI support. This means, for the ARTPEC, that instead of just sending an interrupt to its local MCPU, it can also send MSI messages to a specific address in memory.

As it so happens, we've mapped this area in memory to correspond to the PCIe area. Once something is written to or read from this area, it will propagate across the socket to the other ARTPEC. (Please note that if the ARTPEC driver had been running on the slave, we would also have needed to prevent it from requesting the LCPU IRQ.)

Even though the support is compiled in, the LCPU must be set up to use MSI in addition to traditional interrupts. This is done by sending an MSI setup command struct to the intended subsystem.

Once this is done, it will acknowledge the MSI setup by sending an MSI to the entity which has issued the command. The LCPU will write to a specific memory address which is contained within the memory area detailed in figure 2.15 as *MSI*. A write into this area will make SystemC trigger an interrupt handled by the ARTPEC driver. As such, the master can be alerted of completed commands by the remote LCPU. The size of this specific memory area was made larger than needed, in order to make sure that the following memory areas were page aligned.

MSI Communication

Sending an MSI instead of an interrupt has several advantages when dealing with PCIe end points. For instance, the word written by the MSI in the ARTPEC contains a `vnum`, an `ep` and an unused field as described in figure 2.16. The first two can be used to determine which endpoint (`ep`) and which subsystem (`vnum`) generated the MSI.

```
typedef struct _msi_data {
    unsigned vnum   : 5;
    unsigned ep     : 3;
    unsigned unused : 24;
} msi_data;
```

Figure 2.16: MSI data struct

Using `vnum`, we can establish which remote LCPU subsystem triggered the interrupt after which we can run the appropriate interrupt handler.

Sending LCPU commands

By writing a value to a register representing a FIFO queue in the LCPU, the LCPU is alerted of new commands. Once the LCPU knows which address to fetch the command from (via the value in the register write), it uses DMA to fetch instructions from the MCPU. In our case the MIO will fetch these commands from the remote MCPU's main memory via DMA.

To handle the transfer of commands between the two ARTPECs, we need to place the LCPU command structs in a region mapped by the PCIe driver. An `ioremap` is performed in the ARTPEC driver from the `LCPU_CMD_BASE_ADDR` spanning 64 kB. This gives us a virtual address mapping of the area in which we can place our commands.

Many commands in the driver are, unfortunately, allocated on the stack. As we cannot possibly know where in memory the different kernel threads' stacks are located - we opted for using a bounce buffer. Each issued command is copied to the `ioremap`ped memory before being issued to the LCPU. Once the remote LCPU has completed the command, it will send an MSI and trigger an interrupt handler on the master ARTPEC. The interrupt handler will then copy the completed command back to its original location before triggering any optional callback.

LCPU commands are of varying size, and as such, the amount of memory needed for a command was originally unknown. We handled this by wrapping the LCPU command functions in macros and then including the size in the original function as a parameter as seen in figure 2.17. It is worth noting that the macro will fail if the first argument is a `void` or `char` pointer. As such, the command struct pointer should not be cast to a different type before being passed to this macro.

```
extern int lcpu_cmd_sync_impl(void *, int, size_t);

#define lcpu_cmd_sync(cmd, dest)\
  lcpu_cmd_sync_impl(cmd, dest, sizeof(*cmd))
```

Figure 2.17: Including adding size as a function parameter

As we need to place the structs in a specific memory area, we cannot use convenient allocation functions such as `kmalloc` or `kfree`. Implementing our own free list, arena or buddy allocator would be another risk factor, in an already complex solution.

Fortunately, Linux contains a convenient basic general purpose allocation pool to handle this for us. This pool is located in `genalloc.h`. This pool can be used to manage special purpose memory, and can be told over which area it is allowed to allocate and manage memory. The allocation pool can use best fit or first fit to insert elements. This means that at the cost of performance, it can reduce the amount of fragmentation in the area it manages. It is worth noting that multiple pools can be allocated in the specified area, whereas we are only using one. This is so that we can convert back and forth between virtual and physical addresses.

Since the addresses returned by `genalloc` are virtual, we need to be able to translate these into physical PCIe addresses which the slave LCPU's can address and perform DMA on. Because we are only using one pool from `genalloc`, we are able to perform these translations easily by calculating the offset of an allocation.

```
physical = alloc - pcie_cmd_store + PCIE_LCPU_CMD_BASE_ADDR
```

This address can be written by the MCPU to the LCPU's FIFO queue, as aforementioned. This register write will be sent over PCIe and written directly to the target LCPU. When the write has been completed, the target LCPU can start transferring the command from the supplied address to its own memory using DMA.

Chapter 3

Evaluation

We realized fairly early on, that running the implemented inter-connected system on real hardware would be far-fetched. The sheer amount of issues in the first two months along with fairly untested code led us to only implement the system in Axis's simulated environment.

Two things are good with this approach. First off, the simulated hardware allows for much easier and deeper debugging. This proved paramount to us, since the memory mapping described in figure 2.15, was not trivial to implement in the simulated environment. We had issues with offsets when adding new subsystems and adding support for their different memory channels.

Secondly, the lack of empirical testing forced us to calculate things like bus usage and memory traffic from a theoretical perspective. This helps us predict the best and worst cases for different streaming scenarios.

A drawback with using this approach is the ability to cut corners. It is easy to fake a lot of things using the simulated environment. For instance, one normally has to notify the PCIe controller that MSI data has been read, by writing to an address. We simplify this by reseting the MSI as soon as the PCIe model notices that the MSI has been read, something that is not possible to implement properly in a hardware solution.

Another corner cut, is the fact that reading and writing over the PCIe bridge is not accurately modeled when it comes to latencies. In fact, it is the same latency for writing to a remote subsystem register as to a local one. While in reality, writing over the PCIe bus would incur a penalty.

When calculating theoretical figures for bandwidth et cetera, the real counterpart of these cut corners have been used; so as to establish a realistic assessment of how the implementation would behave on real hardware.

3.1 Results

Internal benchmarks at Axis, show a throughput of roughly 500 MB/s from slave to master over PCIe on the ARTPEC-6, while using a low amount of bandwidth in the opposite direction (about 3 MB/s). The PCIe bridge, however, is supposed to have 700 MB/s on the actual hardware.

The difference in bandwidth between the two directions is explained by the slave being the platform responsible for capturing and processing the images, meaning that the slave's incoming data over PCIe is simply LCPU commands. Once the images have been processed, they are transferred to the master's global memory and sent to the scaling unit (the VPP).

As mentioned in section 2.2.4, there is a flow mode between the image capturing subsystem (MIO) and the image processing subsystem (VIP). This mode can only be enabled when there is a singular video stream being captured by the MIO. If we assume that there are at least two video streams being concurrently captured by the MIO and processed by the ARTPEC, we get the bandwidth savings illustrated by table 3.1 for 1080p and in table 3.2 for 4K, using our dual ARTPEC solution. Note that the table shows the average bandwidth usage per stream between the subsystems and the global memory.

Configuration	Bandwidth/frame (MB)	Bandwidth 60 fps (MB/s)
Single ARTPEC	35.9	2154
Dual ARTPEC (master)	11.9	720
Dual ARTPEC (slave)	26.9	1614
Dual ARTPEC (total)	38.0	2333

Figure 3.1: Average bandwidth usage per stream for 1080p

Configuration	Bandwidth/frame (MB)	Bandwidth 30 fps (MB/s)
Single ARTPEC	142	4273
Dual ARTPEC (master)	48	1425
Dual ARTPEC (slave)	107	3204
Dual ARTPEC (total)	154	4629

Figure 3.2: Average bandwidth usage per stream for 4K

The bandwidth required for a raw image in 1080p and 4K is shown in equations 3.1 and 3.2 respectively. The calculations assume 4:2:0 sub-sampling, resulting in 12 bits per pixel.

$$\begin{gathered} 1920 \cdot 1088 \cdot \frac{12}{8} \cdot 2^{-20} = 2.98828 \text{ (MB / frame)} \\ \Longrightarrow \\ 2.98828 \cdot 60 \approx 179 \text{ (MB / s)} \end{gathered} \tag{3.1}$$

$$\begin{gathered}3840 \cdot 2160 \cdot \frac{12}{8} \cdot 2^{-20} = 11.8652 \text{ (MB / frame)} \\ \Longrightarrow \\ 11.8652 \cdot 30 \approx 356 \text{ (MB / s)}\end{gathered} \tag{3.2}$$

The results in table 3.1 and 3.2 are calculated by using the size of the image being copied to and from different parts of the system. This is explained further by the Python source code in appendix C.

Power consumption

We got a hold of some some rough effect consumption measurements on the actual hardware of the ARTPEC-6. The exact numbers and gradient, however, have been removed to protect Axis's intellectual property. The results of the measurements are illustrated in figure 3.3.

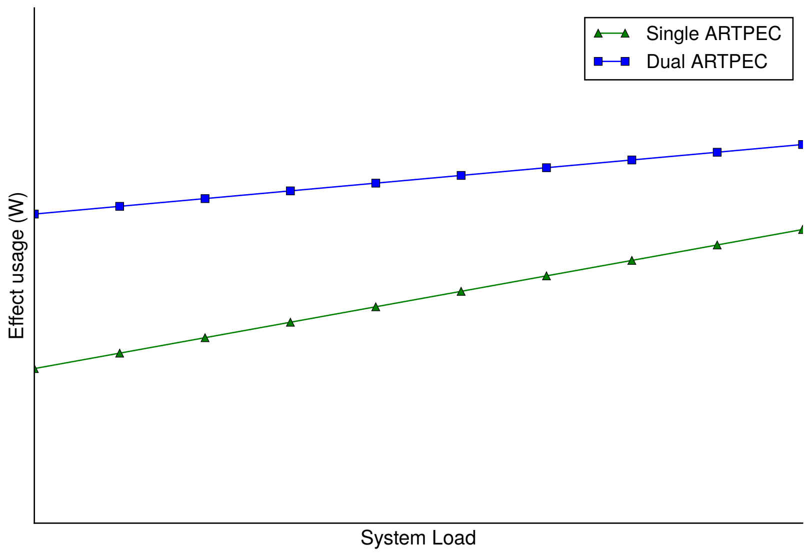


Figure 3.3: Power consumption using single vs dual ARTPECs

3.2 Discussion

3.2.1 Bandwidth

When creating a dual chip solution, we can at best hope to increase the total bandwidth of the system. As shown in table 3.1 and 3.2, the bandwidth usage of the master ARTPEC decreases by roughly 63%, whereas the slave's bandwidth is decreased by roughly 25% for both 1080p and 4K. This assumes that multiple streams are running on the system. If there is only one stream running, the system can make use of the FIFO-mode between the MIO, VIP and VPP. As such, the image needs not be sent to memory until the CDC unit. Effectively making it equally or more efficient than the dual ARTPEC configuration.

The decreases in bandwidth consumption means that the slave will now become the bottleneck. The ARTPEC is able to handle roughly 5.1 GB/s, but unfortunately the PCIe bridge is, as aforementioned, only able to handle 700 MB/s. As shown in equation 3.1 and 3.2, the bandwidth required for a single 4K stream at 30 frames per second requires more than half of the PCIe bridge's bandwidth. The PCIe bandwidth lets us use up to three separate 1080p streams at 60 frames per second.

Amongst Axis's partners, one of the more common scenarios is that one stream is used for live viewing while another, with a different configuration, is used for semi-permanent storage. As such, it is unlikely that two streams with the same configuration will be used. In the single chip configuration, a 4K stream with an additional 1080p stream is impossible. While using the dual chip configuration allows for both streams to be enabled simultaneously.

When calculating the values for the tables in chapter 3.1, the overhead of sending LCPU commands around was ignored. The reason for this is two-fold. First off, the bandwidth required for the LCPU commands is estimated by Axis to be smaller than 5 MB/s. Secondly, this means they do little to impact the performance of the ARTPEC.

3.2.2 Why Choose a Dual Chip Solution?

The performance benefits of using dual ARTPECs over a single ARTPEC has been explained previously. There is also, however, another reason why Axis would choose a dual chip solution over a more powerful single chip solution - namely costs.

Since Axis designs the ARTPEC themselves, it is very inexpensive to produce more chips. This is orthogonal to the way that using third party chip manufacturers works. For instance, Axis does have a product line that uses chip platforms from a company called Ambarella and Axis's current 4K products all use these chips. Since the chips are purchased instead of produced - they are significantly more expensive compared to the cost of printing one of their own chipsets.

In summary, it will be less expensive for Axis to use two ARTPECs compared to using one, more powerful, thirdparty chip. As an added bonus, the costs of maintaining the software will be less if only one type of system is used.

3.2.3 Future Work and Improvements

Modeling Issues

When implementing our model, the first concern was to be able to complete a proof-of-concept before the end of our thesis. This, we have succeeded in. The second goal was to provide Axis with an implementation that would be easy to re-implement on the actual hardware. Ideally, our code would run without modification on the real hardware.

Unfortunately, because of the way we've implemented the PCIe emulation, this is not entirely possible. The PCIe layer has been simplified greatly and there are several issues with this. Firstly, the implementation contains some data that is contained purely within the SystemC model, i.e. not mirrored in Linux. Secondly the slave ARTPEC was not stripped of its operating system, as was Axis's recommendation - on real hardware the slave would not be running an operating system at all. Instead, the PCIe root complex would have to handle the setup of LCPUs on the slave ARTPEC. We think that there is a use case for keeping the slave's Linux OS, which is further discussed below.

ARTPEC Driver Enhancements

When it comes to how LCPU commands are bounced, we're not completely satisfied. It would be much more efficient to not copy every command to a special place in memory; and upon completion, copy them back. Ideally the driver should be modified so that LCPU command allocation is not allowed on the stack or dynamically in kernel memory by using `kmalloc` and the like. Rather, they should only be allowed to be allocated in a specific pre-defined area of memory. This is discussed further in section 3.2.6.

Previous versions of the ARTPEC driver have been proprietary. Thanks to the decoupling between the firmware used in the LCPUs and the ARTPEC driver, Axis has decided to provide the ARTPEC driver with an open source license - the GNU General Public License (GPL). It is this fact that allows us to use structures and functionality from the Linux kernel that are GPL licensed. This includes `genalloc` as well as things like work queues.

Power Consumption

When it comes to the power consumption of built in systems, there is a base effect consumption that is always present [21]. From the effect measurements performed (as shown without hard numbers in figure 3.3), we can conclude that the base leakage and peripheral power consumption will roughly double when using two chips. However, the increase in power consumption in relation to load is somewhat less in the dual chip case compared to the single chip case.

When it comes to the quality of the measurements, at the time of writing, Axis had not performed corner case effect measurements; meaning that the boards that these measurements were performed on, were near perfection. Whereas in later tests Axis will measure worst case and best case performance on a plethora of differently manufactured boards.

As such, the accuracy of the measurements can basically only tell us two things. First off, that the leakage or base effect consumption is linear to the amount of chips used. Second off, they tell us that the slope of the curve is decreased when using two chips. In figure 3.3 it looks like they are converging quite fast, but the truth is that they might not converge that fast or at all. This is due to the fact that there will be a theoretical max load that the chip is able to handle, after which the power consumption will be constant.

Utilizing The Slave MCPU

As previously mentioned, our current implementation allows the slave system to use Linux in order to set up the firmware for the LCPU subsystems. This means that while the slave has an MCPU, this unit is not used for anything after the setup and will, de facto, be idle.

The same goes for the remaining subsystems not being used on the slave. Namely the VPP, CDC, Crypto and various other peripherals.

We believe that some tasks could be offloaded to the slave MCPU without effecting the bandwidth limitations of the PCIe bridge.

One possible application is audio capture and processing. Since the slave already performs the image capture, it might not be a bad idea to also let it perform the audio capture and processing. In current Axis products, the audio resampling is done with emphasis on speed rather than quality. This is because the resampling is performed on the MCPU and it is quite CPU intensive to do well. In the dual ARTPEC case, this resampling could be offloaded to the slave's MCPU.

This use case provides a basis for why Axis should not strip the slave of its Linux OS, which is further discussed below. While this example might be trivial, the main focus of our implementation is to boost the video processing power of upcoming ARTPEC products. Thus, any utilization of the slave MCPU may not interfere with the PCIe link's bandwidth limitation between the two ARTPEC chips. Audio processing would likely take up no more than 5 Mbit/s across the PCIe bridge, which is small enough not to exacerbate the performance.

Removing The Slave's OS

During the information gathering portion of the thesis, we discussed with several different people at Axis what to do with the slave's OS. The consensus at Axis was that the slave's OS should be removed. This was motivated by the fact that having two operating systems would complicate the resulting system.

In our implementation we chose to keep a stripped slave OS for the simple reason that it would allow us to skip implementing the firmware initiation of the remote LCPUs. We did - however - remove the ARTPEC driver. This driver would have generated conflicts and perhaps issues that would have been hard to debug.

While we do agree, that having two independent systems is overly complicated; we think that the use of the same type of API that is used for the LCPUs, can be leveraged to make great use of the slave's MCPU and idle subsystems.

In our opinion, one of the most impressing features of the ARTPEC is the modularity on the SoC. Each subsystem can be controlled remotely, and does not depend on peripherals or mutable state. They can be used in a declarative way; a command struct specifies what to do, where to get the relevant data from and where to place any results.

This model could be transferred to being used on the slave MCPU that could act as its own subsystem, a Slave CPU (SCPU). The SCPU would work in the same way as the other subsystems, a command struct would be sent to a specific FIFO address - it would perform calculations and emit the resulting data to the specified address.

In the case of offloading the audio resampling to the SCPU, the master ARTPEC could send a command to the slave system, saying *capture audio and resample it, then write it to this address*. The resulting audio stream could then be muxed together with the video stream in the master without much hassle. This also means that Axis would be able to offer higher quality audio - without needing to invest in specialized hardware like digital signal processors.

To do this, however, the system needs access to an audio infrastructure. Much of this infrastructure is provided by the Linux kernel with extensions available in specialized programs for audio processing. A bare metal system would not be able to handle this quite as easily as a system running Linux.

When it comes to power limitations, the MCPU can be loaded pretty heavily without affecting the effect consumption. Thus, if the system is already a dual chip solution, then it will not matter, power-wise, if the slave is running its own OS or not.

Limitations of Flow

As previously established, there is a single direct data channel across the MIO, VIP and VPP. After the VPP there are multiple direct data channels that can be directed to memory, and subsequently sent to the CDC for encoding.

In the scenario of a singular stream, our dual ARTPEC solution is not beneficial. This is because the data won't enter main memory before the coding unit. As such, the utilized bandwidth will be the same on a single as dual ARTPECs (disregarding the bandwidth for transferring over PCIe).

Flow provides the possibility to process a single image stream throughout the ARTPEC until the VPP, but generate multiple streams at different resolutions. In this scenario the benefit is also moot, because of the same reason as for a singular stream.

The real benefit is when there are multiple sensors attached to the ARTPEC. In this scenario, flow between the MIO and VIP cannot be used. As such, the need to bounce the image in memory becomes mandatory. The benefits of running this scenario on dual ARTPECs is the one illustrated by table 3.1 and table 3.2.

Multiple ARTPECs

As the thesis title suggests, it is indeed possible to use more than one slave ARTPEC to create a multi-ARTPEC solution. The hardware supports this, and Axis has even performed tests in which a master ARTPEC is able to address more than one slave over PCIe. This is done by using a switch between the different endpoints from the root complex.

With that being said, the main question is - does our implementation support multiple ARTPEC slaves? The answer is that it does - sort of. This is due to the fact that while the hardware and driver support is there - we simply see no real use for multiple slaves. Our method of dividing the work would not improve from having more slaves available. This is because the limiting factor in our implementation is the PCIe bridge.

The ability to have multiple endpoints, however, is great - why hook up an ARTPEC? You could hook up a totally different SoC that provides functionality that the ARTPEC does not.

3.2.4 Issues with QEMU

Interrupt Numbers

The first problem that we encountered when implementing the simulated PCIe bridge was the indexing of the interrupt numbers. It turned out that the interrupt numbers created in QEMU are not mapped to the same numbers as the Linux guest operating system. After reading a vague comment about it in the source code [12], we found out that there is a constant offset of 32. The reason being that the Cortex-A9MP chip numbers external interrupts starting from 32 [10, Chapter 3.12], where our PCIe bridge is to be considered an external device, and therefore emits external interrupts.

Alignment and Data Transfers

When sending the data over the simulated PCIe bridge, there are a few things to consider. There are a total of four (possibly different) operating systems involved when sending data over the socket, one guest and one host per machine as illustrated in figure 2.9. In order to send data over the simulated PCIe, the dataflow is as seen in figure 3.4.

1. Guest 1 writes the command struct to the I/O mapped memory in QEMU, this is 4-byte aligned.
2. Host 1 wraps the command struct in a header struct and writes to the socket.
3. Host 2 reads the data from the socket and interprets it as a header struct.
4. Guest 2 reads from the I/O mapped memory, again 4-byte aligned, and interprets this as a command struct.

Figure 3.4: Data flow

At each read and write, it is important that the different operating systems interprets the bytes in the same way. The compiler of a C program is allowed to add padding to the various fields in a `struct`, either because it is required by the architecture, or because it results in a performance gain [4]. If the two hosts, or the guests, or both, add padding in a different way, the interpretation will differ and the communication is broken. The GCC [3] compiler used supports specifying attributes of types [14], where the two interesting attributes are *packed* and *aligned*. Here, *packed* effectively removes padding as much as possible, and *aligned* aligns the struct and its members to a specific minimum alignment.

The first attempt without any special attributes resulted in that the QEMU drivers could talk to each other and correctly read the messages, but the guest operating system could not read the received bytes properly. An attempt was made where we used the *packed* attribute, but this still resulted in the same error, possibly because of the 4-byte alignment when reading from the I/O mapped memory. Finally *aligned* with a 32-byte alignment fixed all the issues, this results in an increased memory usage, but it is the same alignment as required by the LCPUs.

It is also worth noting that QEMU does not allow byte level access to data, but rather chunks the read and writes to a `uint32_t` or word level. This can be seen in the function declarations in figure 2.11.

Threading Issues

As previously mentioned, QEMU is wrapped as a library and uses one instance for each CPU in TLMu. This means that the code in QEMU will be run by the SystemC instance. Since SystemC does not support concurrently running threads, but rather allows one thread to run at a time. It also imposes restrictions on when memory can be read or written.

In our solution using QEMU, we tried to let a `pthread` continuously read from a socket and perform memory IO by calling `cpu_physical_memory_read` or `cpu_physical_memory_write` when applicable; this was a bad idea. SystemC immediately crashes when QEMU tries to write to the physical memory outside of IO functions. This is because SystemC does not allow writes to memory outside of `SC_THREAD`s.

This is not surprising, as SystemC aims to really emulate how hardware behaves; it should impose these types of restrictions. Unfortunately, we could not hack around this issue and were forced to abandon this implementation strategy.

3.2.5 Issues with TLMu

Port and Socket Issues

There were a couple of problems that occurred when using QEMU. First off, the version used in the TLMu environment is a very old version, dated back to 2010-08-16 [9]. The documentation is fine when using the application, but when we were required to program in QEMU it was hard to find proper documentation, and we often had to go through the source code in order to gain information.

Another problem with this environment is that it is not possible to run two TLMu instances on the same computer. This is due to the fact that there are a lot of hard-coded ports and sockets used as means for QEMU to communicate with SystemC as well as the various help utilities. Since the environment models a complete ARTPEC a lot of utilities are used to emulate hardware implemented features, e.g. H264 coding. As such, both the TLMu implementation as well as the various utilities would need to be re-configured to use different ports, to allow multiple instances on one computer.

An attempt was made to reconfigure all the ports, but it still resulted in crashes. In the end we had one TLMu instance per computer and the workflow was fine.

Message Passing Between ARTPECs

Implementation wise, the message passing between machines, is the same for both QEMU and SystemC. The first machine sends a message and the other machine acknowledges that it has received the message by echoing the sequence number.

Since, however, the SystemC implementation dynamically decides if it is the endpoint or the root complex - the complexity is higher in this implementation. The reason for choosing this approach, was that the workflow in building and starting the two simulations would be lighter considering the trial and error style programming employed. In retrospect, a static decision would have been easier to deal with - however, the lack of testability on a unit level would not have alleviated the trial and error style programming.

One of the bugs encountered because of the dynamic decision making was sending an `ACK` on every message. This was later refactored so that an `ACK` would always be sent else an error would be generated.

Slow Transfer Rates Over PCIe

As aforementioned, one of the issues with trying to implement the socket connection in QEMU is the fact that we're not allowed to block in SystemC threads. As such, when we switched to implementing the socket connection in SystemC, we had to use SystemC threads without blocking. Because of this, we chose to implement the socket communication using polling on both sides.

The waiting thread would call SystemC's `wait` function if no data was available, thus yielding execution to the next SystemC thread. The problem with this approach is that the resulting transfer rate is terrible. Despite decreasing the wait time, the transfer rate only increased slightly - however, since the other threads are allowed less time to progress - the whole simulation slows. From what we understand, this is due to the fact that SystemC does not actually use *real* threading. Instead it lets each SystemC thread run, one after the other in the SystemC context. Instead of letting each thread run concurrently on the host computer's CPU.

In IEEE 1666-2011, SystemC introduced `async_request_update`, which allows non SystemC threads to update the simulation [5]. Once this feature was discovered, we could re-implement the socket connection using POSIX threads, allowing asynchronous update calls to the simulation exactly when data is available. This improves the data transfer immensely, allowing us to pass larger images over PCIe in a time frame that is seconds as opposed to hours.

The second problem which occurred when large amounts of data had to be transmitted, was the fact the DMI could not make use of DMA. This occurs when transmitting frames from the process buffers on the slave to the VPP subsystem on the master. Each time 72 KB had to be transmitted (the largest amount of data transmitted in the simulator), 4 byte transfers were issued to QEMU. This resulted in low FPS in the simulator and a large overhead in the PCIe model.

Each PCIe packet, excluding the payload of 4 bytes, requires 9 bytes (11 bytes including possible padding). This means that in order to transfer 72 KB over PCIe, 18432 packets over PCIe needed to be transmitted, each one causing a large overhead in the simulator.

First a new xbar was added in order to bypass the platform, enabling DMA to be used with PCIe, which in turn made it possible to make memory accesses much larger than 4 bytes (in practice 72 KB accesses were made). This forced us to change the way PCIe packets were transmitted, and we had to make sure that PCIe packets could carry a dynamic amount of data. Finally 72 KB transfers over the Unix socket were possible with an overhead of only 16 bytes (no padding) and the performance thus increased.

3.2.6 Handling LCPU Command Results

The LCPU commands will be transferred back to their origin address via DMA - this means that the PCIe back end must be able to facilitate this in some way.

As mentioned section 2.4.5, we chose to avoid this problem by copying the LCPU commands to a bounce buffer addressable from the memory area mapped to PCIe.

This, unfortunately, is not the most efficient implementation. There are, however, two efficient ways of implementing this that were considered. The first way was to let the PCIe back end map the entire kernel stack (for all kernel threads). With this approach, we would not need to copy the results back and forth - the slave could transfer the structs back via DMA directly. Since we cannot be sure where Linux will decide to allocate its kernel thread stacks - we would need to map the entire memory available, which we cannot do with PCIe. The ARTPEC-6 is limited to 512 MB of mapped PCIe addresses, making it impossible to make sure that all memory used by Linux is covered; since the ARTPEC-6 will have more than 1 GB global memory.

Another alternative to mapping the entire kernel stack space would be to simply allocate the messages in a specific part of memory `ioremap`ped by the PCIe backend. This approach was, however, not chosen because it would mean having to change a lot of code in the driver.

An issue with the bounce buffer approach was that to copy the structs back to memory, we needed to know the size of the command structs. When copying to the PCIe memory area, however, we could have simply copied a large enough chunk of memory to cover the command struct.

Knowing the size of these structs was not possible in the original ARTPEC driver. We introduced the macro detailed in figure 2.17 to fix this, however, the macro has some down sides. It cannot be passed around as a function pointer and the user has to be aware of the fact that it is a macro, and therefore has these limitations, as mentioned in section 2.4.5.

One problem with the bounce buffer approach is that the memory consumed by the ARTPEC driver is increased by the amount of LCPU commands concurrently running. It is also possible that synchronous commands will time out and if they are stack allocated - get popped. If a command times out but is written back to the stack after timeout, the DMA transfer might overwrite stack memory that is being used for something else. As such, implementing the solution with a dedicated memory area would be beneficial for both memory usage and stability.

To implement the memory area, using something like `genalloc` is entirely plausible. This pool can be tweaked using features like best-fit instead of first-fit, as such the choice can be made between increased speed or better memory utilization. Arenas are another alternative, but the amount of different LCPU commands would make this approach fairly tedious. Each arena would store commands of the same size, and to know how large each arena should be - a more thorough analysis of the ARTPEC driver would be needed.

3.2.7 Direction of PCIe Accesses

In order to maximize the performance, we have to consider how we interface with the PCIe device; in particular we have to consider if we can use writes instead of reads. The first optimization is the location of the buffers, in this case the process buffer. The VIP writes to the slave's global memory, and then the master reads from it, the process buffer should therefore be placed in the master's global memory. One could argue that the same thing applies for the command structs, that the ARTPEC UDL driver should have two memory pools, one on the master - and one on the slave side where the commands are bounced. However, when the command has finished, the driver must fetch the command struct back to master. The ARTPEC UDL driver does not use DMA - something that the LCPU's always do - and therefore the fetch would require multiple 4 byte reads over PCIe.

3.2.8 Alternative Implementation

When looking back at our implementation across two separate instances of TLMu and SystemC models, we've come to the conclusion that we could have implemented two ARTPECs using one SystemC simulation. For the initial stage of the implementation this would have been a smoother approach, especially since we had so many issues with the sockets and threading. However, it would not have entirely solved our problem with the internal TLMu socket communication for CPUs - these would have clashed regardless.

3.2.9 Development Workflow

When making changes to the ARTPEC kernel driver, a simple rebuilding of the kernel image followed by rebooting the simulation constitutes the actions needed to reflect the changes in the simulation. This needs only be done on the ARTPEC master simulation, thus changes in the ARTPEC driver are preferable to those made in the simulation.

When developing functionality in the SystemC simulation however, the whole project needs to be compiled on both ends. The changes were usually made on the master host, and then pushed to Axis's git repository, followed by a pull on the slave host. When we were developing in QEMU, the whole SystemC project had to be recompiled, which is very time consuming unless something like `ccache` is used [2]. This is because SystemC is a large project written in C++.

As such, once we had the simulation working properly, the whole workflow improved drastically.

Looking back at the software written during this thesis, testability has been a huge issue. We both come from a background where test-driven development (TDD) plays a huge roll in development. As such, the absence of unit, integration and regression tests in the simulation and drivers has been somewhat new to us. Testing is instead performed by external scripts or by using GDB to make sure that the correct values are passed on.

3.3 Conclusion

We think that Axis has a real winner on their hands with the dual ARTPEC concept. Thanks to ingenuity of the subsystem structure and the ARTPEC driver, the complexity of the dual ARTPEC lies mainly in the sharing of LCPU command structs and efficiently implemented buffer sharing.

We have shown that the resulting system reduces both systems' bandwidth usage; the master ARTPEC's bandwidth usage is decreased by 63% while the slave's usage is decreased by roughly 25%. The dual chip implementation also provides other opportunities, such as the ability to use the slave's CPU as a separate subsystem.

We strongly advocate for the usage and development of applications for the slave's MCPU - as it would otherwise stand idle. As the PCIe bridge becomes the choke point, applications offloaded to the slave have to take care not to transmit data across the bridge unnecessarily.

The resulting implementation provided by this thesis has taken some shortcuts. This is due to the fact that the implementation was performed on simulated hardware as opposed to real hardware. This means that the implementation in itself does not map directly to hardware - but it is close enough that the real implementation should very be straightforward.

Bibliography

[1] Axis Developer Wiki - FAQ. http://developer.axis.com/wiki/doku.php%3Fid=faq.html. Accessed: 2016-02-16.

[2] ccache. https://ccache.samba.org/. Accessed: 2016-02-23.

[3] GCC, the GNU Compiler Collection homepage. https://gcc.gnu.org. Accessed: 2015-11-18.

[4] IBM developerWorks data alignment straighten up and fly right. http://www.ibm.com/developerworks/library/pa-dalign. Accessed: 2015-11-18.

[5] IEEE 1666-2011. http://standards.ieee.org/getieee/1666/download/1666-2011.pdf. Accessed: 2016-01-05.

[6] ITU h.264 : Advanced video coding for generic audiovisual services. http://www.itu.int/rec/T-REC-H.264-201402-I/en. Accessed: 2015-11-03.

[7] Linux Kernel Virtual Machine. http://www.linux-kvm.org. Accessed: 2015-12-01.

[8] QEMU open source processor emulator. http://wiki.qemu.org/Main_Page. Accessed: 2015-11-02.

[9] qemu.git short log of tag 0.12. http://git.qemu.org/?p=qemu.git;a=shortlog;h=refs/heads/stable-0.12. Accessed: 2015-11-18.

[10] RealView Platform Baseboard Explore for Cortex-A9 User Guide. http://infocenter.arm.com/help/topic/com.arm.doc.dui0440b/DUI0440B_realview_platform_baseboard_for_cortexa9_ug.pdf. Accessed: 2015-11-26.

[11] The PCI Express Port Bus Driver Guide HOWTO. https://www.kernel.org/doc/Documentation/PCI/PCIEBUS-HOWTO.txt. Accessed: 2016-02-17.

[12] TLMu GitHub tlmu/hw/vexpress.c. https://github.com/edgarigl/tlmu/blob/d60ca691992e9f987e9c855ae5fb04b1730468f1/hw/vexpress.c#L106. Accessed: 2015-11-18.

[13] Understanding Performance of PCI Express Systems. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf. Accessed: 2016-02-04.

[14] Using the GNU Compiler Collection (GCC) specifying attributes of types. https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/Type-Attributes.html. Accessed: 2015-11-18.

[15] P.R. Bashford. Message signaled interrupt generating device and method, September 30 2003. US Patent 6,629,179.

[16] Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.

[17] PCI Special Interest Group. PCI Express Base Specification Revision 3.0. 2010.

[18] Edgar E. Iglesias. Tlmu. 2011.

[19] Open SystemC Initiative et al. IEEE standard SystemC language reference manual. *IEEE Computer Society*, 2006.

[20] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers - 3rd Edition*. O'Reilly Media, 2005.

[21] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *computer*, 36(12):68–75, 2003.

[22] Sudeep Pasricha. Transaction level modeling of soc with systemc 2.0. In *Synopsys User Group Conference (SNUG)*, volume 3, page 3, 2002.

[23] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor soc platform applied to networking and multimedia. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7):667–680, July 2006.

[24] H. Linse R. Lindahl. Real-time Panorama Stiching using a Single PTZ-Camera without using Image Feature Matching. 2015.

[25] Marco Sironi and Francesco Tisato. Capturing information flows inside android and qemu environments. *arXiv preprint arXiv:1302.5109*, 2013.

[26] A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl. System level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms. *IEE Proceedings-Computers and Digital Techniques*, 152(1):3 – 11, 2005.

[27] Yin Yaming and Chen Shuming. Design and implementation of a inter-chip bridge in a multi-core soc. National University of Defense Technology, Department of Computer Science and Tehnology, Changsha, 410073, China, 2009.

[28] Zhaohui Zhong, Deli Wang, Yi Cui, Marc W Bockrath, and Charles M Lieber. Nanowire crossbar arrays as address decoders for integrated nanosystems. *Science*, 302(5649):1377–1379, 2003.

Appendices

Appendix A
Glossary

ARTPEC	Axis RealTime Picture Encoding Chip
CDC	Coding and Decoding Unit
CPU	Central Processing Unit
DMA	Direct Memory Access
DMI	Direct Memory Interface
DRAM	Dynamic Random-Access Memory
EP	Endpoint
FIFO	First-In-First-Out
GPL	GNU Public License
Global Memory	the main memory of the ARTPEC addressable from the MCPU
I-frame	Inter frame
IPP	Image Processing Pipeline
IRQ	Interrupt Request
KVM	Kernel-based Virtual Machine
LCPU	Local CPU
Linux	a Unix-like and mostly POSIX-compliant computer operating system
MCPU	Main Central Processing Unit
MIO	Memory Input/Output
MSI	Message Signaled Interrupts
P-frame	Predicted frame
PCIe	Peripheral Component Interconnect Express
QEMU	a generic and open source machine emulator and virtualizer
RAM	Random-Access Memory
RC	Root Complex
SCPU	Slave CPU
SRAM	Static Random-Access Memory
Sensor	an input to the MIO, usually a camera lens and sensor

SoC	System-on-Chip
SystemC	A C++ framework for event-driven simulation
TDD	Test-driven Development
TLM	Transaction Level Modeling
TLMu	Transaction Level Emulator
UDL	User Defined Logic
VIP	Video Input Processor
VPP	Video Post Processor

Appendix B
ARTPEC-6 Usage

```c
int main(void) {
  /* Open device */
  int fd = open("/dev/cam0", O_RDWR);

  /* Add a new stream with the specified configuration */
  ioctl(fd, CAM_IOC_STREAM_ADD, stream_config);

  /* Start capture */
  ioctl(fd, CAM_IOC_STREAM_CONTROL, 1);

  /* Allocates one struct encode_request */
  int enc_phys = ioctl(fd, CAM_IOC_ENCODE_ADD, IMAGE_JPEG);

  /* Map encode request to user space */
  struct encode_request* enc =
    mmap(NULL, sizeof(*enc), PROT_READ | PROT_WRITE, MAP_SHARED, fd, enc_phys);

  /* Start encode process */
  ioctl(fd, CAM_IOC_ENCODE, phys);

  /* Wait for encode to finish */
  select(fd, NULL, NULL, NULL, NULL);

  /* Encoded frame is now available in `enc` */
  return 0;
}
```

Appendix C
Bandwidth Calculations

```python
#!/usr/bin/env python3

# Calculate bandwidth using the following image dimensions:
width  = 1920
height = 1088
fps    = 60

# Calculate bytes per image:
pixel_size = 12
byte       = 8.0
raw_size   = width * height * pixel_size / byte
raw_meta   = width * height

# H264 encoding for 30 fps:
h264_30_fps = 10 * 1024 * 1024 / byte # 10 mbit/s to MB/s
h264_size   = h264_30_fps / 30

# Define convenience functions:
def to_kb(from_bytes):
    return from_bytes / 1024.0

def to_mb(from_bytes):
    return to_kb(from_bytes) / 1024.0

def print_res(from_scenario, with_bytes):
    mb = to_mb(with_bytes)
    print("Results from: {}".format(from_scenario))
    print("{0:10.3f} MB/image".format(mb))
    print("{0:10.3f} MB/s at {1}x{2} @ {3} fps\n".format(mb * fps, width, height, fps))


################################################################################
#                                                                              #
# Scen 1: single chip, multiple streams, calculate bw needed for one stream    #
#                                                                              #
################################################################################

def scen1():
    # Between MIO => DDR
    mio_to_ddr = raw_size
    # Between VIP <=> DDR, assuming a simplified IPP
    vip_to_ddr = raw_size + raw_meta * 2 + \
                raw_meta + raw_size * 4
    # Between DDR => DDR (to VPP/CDC buffer)
    ddr_to_ddr = raw_size
    # Between VPP <=> DDR
    vpp_to_ddr = 2 * raw_size
    # Between DDR => CDC
    ddr_to_cdc = raw_size
    # From CDC => DDR (Completed image)
    cdc_to_ddr = h264_size

    return mio_to_ddr +\
           vip_to_ddr +\
           ddr_to_ddr +\
           vpp_to_ddr +\
           ddr_to_cdc +\
           cdc_to_ddr

################################################################################
#                                                                              #
# Scen 2m: dual chips, multiple streams, calculate bw needed for one stream,   #
#          on master ARPTEC                                                    #
#                                                                              #
################################################################################

def scen2m():
    # From slave over PCIe to DDR
    pci_to_ddr = raw_size
    # Between VPP <=> DDR
    vpp_to_ddr = 2 * raw_size
    # Between DDR => CDC
    ddr_to_cdc = raw_size
    # From CDC => DDR (Completed image)
    cdc_to_ddr = h264_size

    return pci_to_ddr +\
           vpp_to_ddr +\
           ddr_to_cdc +\
           cdc_to_ddr

################################################################################
#                                                                              #
# Scen 2s: dual chips, multiple streams, calculate bw needed for one stream,   #
#          on slave ARPTEC                                                     #
#                                                                              #
################################################################################

def scen2s():
    # Between MIO => DDR
    mio_to_ddr = raw_size
    # Between VIP <=> DDR, assuming a simplified IPP
    vip_to_ddr = raw_size + raw_meta * 2 + \
                 raw_meta + raw_size * 4

    # To master via PCIe
    pci_to_mas = raw_size

    return mio_to_ddr +\
           vip_to_ddr +\
           pci_to_mas

print_res("Scenario 1", scen1())
print_res("Scenario 2 (master)", scen2m())
print_res("Scenario 2 (slave)", scen2s())
print_res("Scenario 2 (combined)", scen2m() + scen2s())

print("Results are missing overhead from LCPU commands amongst other things")
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-03-18

EXAMENSARBETE Image Processing Across Multiple Interconnected System-on-Chips
STUDENT Andrée Ekroth, Felix Mulder
HANDLEDARE Jonas Skeppstedt (LTH), Johan Rudholm (Axis), Magnus Mårtensson (Axis)
EXAMINATOR Per Andersson

Image Processing Across Multiple Interconnected System-on-Chips

POPULÄRVETENSKAPLIG SAMMANFATTNING **Andrée Ekroth, Felix Mulder**

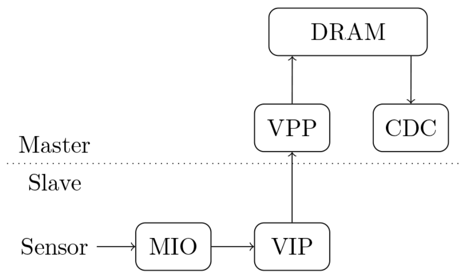
This theis examines how best to divide the work of image processing across multiple interconnected System-on-Chips and shows by proof-of-concept that it is a feasible solution for bandwidth demanding applications.

Splitting Up The Work
The motivation for the thesis was to allow Axis to utilize the hardware they designed themselves - instead of purchasing third party chips - to allow for high bandwidth applications like streaming at high resolutions.

During our thesis work, we have implemented a solution for dividing work between two SoC:s interconnected using a PCI Express (PCIe) bridge. It is easiest to imagine the relation between the two chips as a slave and master relationship, where the master simply tells the slave what to do.

Each chip contains several subsystems, and the good people at Axis have implemented an API to communicate with these subsystems. The API basically uses command structures sent from the operating system on the CPU to the subsystems containing Local CPUs (LCPUs). The beauty of this approach is that the system becomes very decoupled and modular - i.e. each system operates independently of the others.

In our thesis, we have split the work for different subsystems across two ARTPECs. In the single ARTPEC case, all subsystems are on the same chip.



Basically the figure illustrates the images' path through the different subsystems to the point at which it is made available for user consumption. To start with, we feed the system with a known image at the sensor stage on the slave side. When the image has been captured and processed by image improvement algorithms, it is sent via PCIe to the master ARTPEC. At this point the image is scaled and then sent for encoding. Once these steps have been performed, the image can be delivered to the user application.

We have chosen to divide the work by placing the image processing on the slave and the post-processing on the master ARTPEC. This is because the most bandwidth demanding subsystems are the image processing (VIP) and post-processing (VPP).

Results and Future Work
Our implementation results in a significant decrease of memory bandwidth usage on both the slave and the master ARTPEC. Most significantly it decreases the slave's bandwidth usage with 63% and the master's with roughly 25%.

Both the master and the slave ARTPEC contain all subsystems. This means that with our implementation the slave has subsystems and, perhaps more importantly, an entire CPU that is not being used. What this entails is that we have effectively enabled offloading of strenuous tasks!

In order to offload tasks to remote subsystems, it would be very easy for Axis to write an API that works in the same way as their current subsystem control mecha-nism. Examples of things they could easily offload are things like: audio re-sampling, user applications dubbed ACAP, facial recognition, and object recognition to name a few.