EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Motion controller acceleration by FPGA co-processing

van Broekhoven, M.

Award date:
2014

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Publication Date: September 29, 2014

2IM91

Motion Controller Acceleration by FPGA Co-Processing

M. van Broekhoven
(0653605)
m.v.broekhoven@student.tue.nl

University Supervisor
dr. R.H. Mak
r.h.mak@tue.nl

Prodrive Supervisor
E. van Uden
eric.van.uden@prodrive-technologies.nl

September 5, 2014

Version: 1.1

Contents

1 Introduction **4**

1.1 Photolithography Machines 4

1.2 Challenges for Future Servo Applications 5

1.3 Problem Statement 6

1.4 Solution Direction 7

1.5 Document Structure 7

2 Background Information **9**

2.1 CARM Software Architecture 9

2.1.1 Terminology Overview 11

2.1.2 Implementation 12

2.1.3 Execution Scenarios 17

2.2 CARM Hardware Architecture 20

2.2.1 ATCA Racks 20

2.2.2 HPPC 20

2.2.3 RapidIO Interconnect 22

2.3 Benchmark Application 23

2.3.1 State Space Calculation 23

2.4 Previous Work 24

2.4.1 Dataflow-based Multi-ASIP Platform Approach for Digital Control Applications 24

2.4.2 A Design-Space Exploration for High-Performance Motion Control . . . 26

3 Analysis and Approach **27**

3.1 Block Calculation 27

3.2 Slack Estimation 28

3.3 Control Mode Switches 29

3.4 State Control 33

3.5 Hardware Architecture 33

3.5.1 Solution Options 34

3.6 Sample Frequency Feasibility Analysis 34

3.7 Functional Overview 37

4 MCCP Proof of Concept Interface **40**

4.1 Programming the ASIP Platform 40
4.2 MCCP Configuration 41
4.3 Task Input 41
4.4 Task Output 42
4.5 Parameter Set Loading 43

5 MCCP Proof of Concept Design **45**

6 Experimental Evaluation **50**

6.1 Test Setup 50
6.2 Software 50
6.3 Benchmarks and Metrics 51
6.3.1 Delays 53
6.4 Internal Benchmarks Approach 53
6.4.1 Task Input-to-Output Delay 53
6.4.2 PSU Parameter Load Delay 54
6.4.3 FPGA Resource Utilisation 55
6.4.4 External Benchmarks Approach 55
6.5 HPPC Simulator Benchmarks Approach 55
6.5.1 Correctness 56
6.5.2 Delay 56
6.6 Results and Analysis 57
6.6.1 Resource Utilisation 57
6.6.2 MCCP Timing Metrics 58
6.6.3 HPPC Simulator Timing Metrics 60
6.7 Experimental Evaluation Conclusion 62

7 Conclusion and Future Work **64**

A List of Abbreviations **66**

B Resource Utilisation **67**

1 | Introduction

Prodrive Technologies serves a wide market in making technology solutions. ASML, one of the world's leading designers and manufacturers of lithography machines (Section 1.1), is one of its customers. Over the years, several ASML projects have been outsourced to Prodrive Technologies. This graduation project is part of one of those outsourced projects: the CARM motion controller project. The CARM motion controller project is the project which develops the framework for motion controllers based on [1]. It is explained in Section 1.1.

1.1 Photolithography Machines

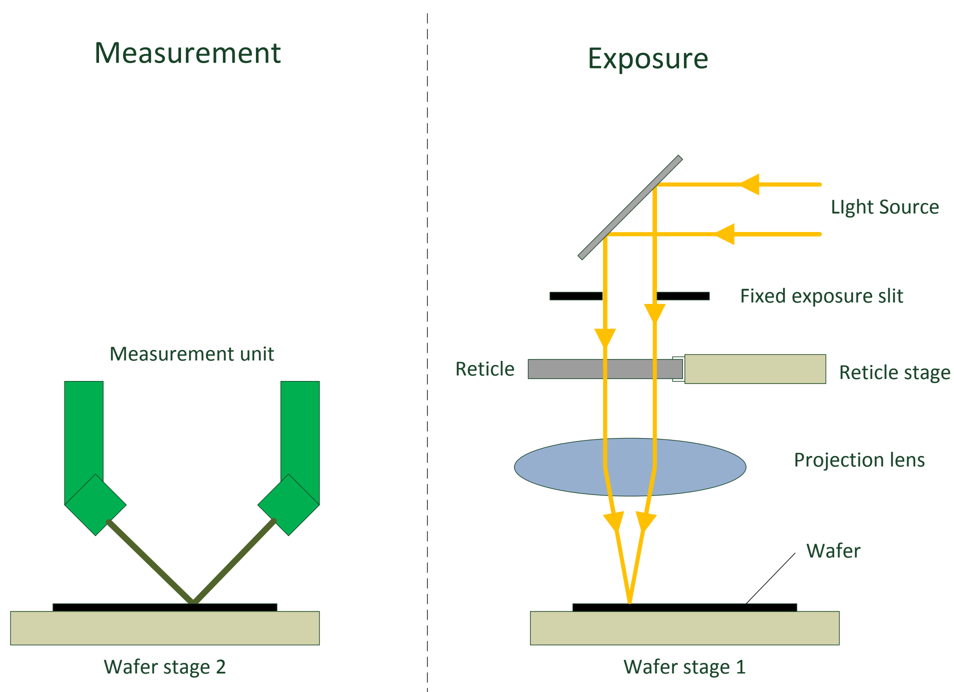


Figure 1.1: Dual-Stage Waferscanner

Lithography machines like the Dual-Stage Waferscanner (Figure 1.1) are used to manufacture integrated circuits (ICs). ICs are created from discs of silicon called wafers. Wafers are treated with photo-resist material before being fed into the Waferscanner. The Waferscanner then exposes the wafers to patterns of light. During exposure, the Wafer stage and Reticle stage move independently of (but not unrelated to) each other. The accuracy of these movements has to be very high in order to avoid creating errors on the ICs. The necessary accuracy

is dependent on the minimum feature size (smallest used element) of the integrated circuit; the minimum feature size is inversely proportional to the necessary accuracy. At the same time, the machine has to work very fast (i.e. be able to process a high amount of Wafers per hour) in order to maintain a competitive advantage in the photolithography machine market. As described in [2], one technique used in the Dual-Stage Waferscanner, is to use two Wafer stages in one photolithography machine. One wafer stage is used to measure one wafer's surface while at the same time another wafer is exposed. When both exposure and measurement is finished, the two stages switch position, or the wafers are moved out of the machine.

A motion controller controls the motion of a physical object, such as the wafer stage in the Waferscanner. This motion controller periodically receives sensor inputs. With the sensor inputs, it computes outputs used to actuate the controlled object. For example, a position sensor input could be used to calculate the position error and compensate for this calculated error with its actuator outputs. The software that implements a motion controller is called a Servo application (Figure 1.2). Important metrics of a Servo application are

- Sample frequency: the frequency at which both sensor inputs are sampled and servo outputs are computed
- IO-latency: the time it takes from the moment sensor inputs are consumed to the moment the servo outputs have been computed and appear on the outputs

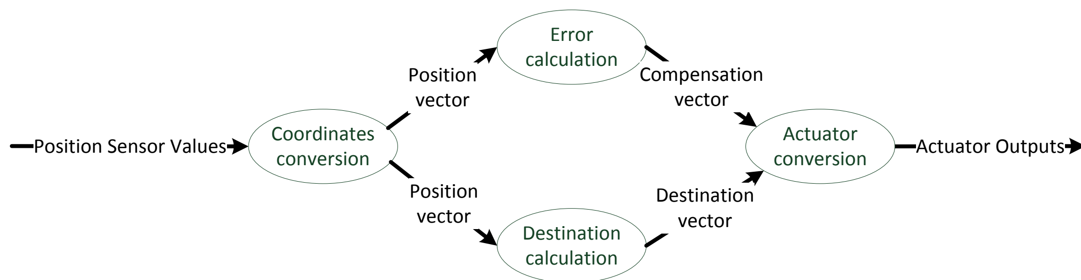


Figure 1.2: Example of a servo application.

A CARM servo application can execute in different modes. For example, during measurement, the wafer stage need not move as accurate as during exposure. As a result, the servo application of the wafer stage has more slack in the accuracy of its actuator outputs and can therefore move faster during measurement than during exposure. These different modes of execution are set by supervisory control.

1.2 Challenges for Future Servo Applications

Engineers aim to maintain Moore's law: the amount of transistors on a chip has to be roughly doubled every two years. To achieve this, the minimum feature size of the transistors on chips is decreased. Because of smaller feature sizes, future generations of photolithography machines have to take into account much smaller vibrations and disturbances than currently play a role.

An example of these small vibrations are non-rigid body vibrations. Non-rigid body vibrations typically have a high frequency, hence in order to compensate for them, measurement samples have to be taken at an even higher frequency. In addition, compensating for these small vibrations is more computationally intensive. The increase in both sample frequency and intensity of computation greatly increases the computational load on motion controllers. To be able to implement servo groups with non-rigid body vibration compensation, there is a long term goal of a sample frequency of 100kHz with an IO latency of 10μs for future servo groups.

Currently, ASML uses a general purpose multi-core multi-processor platform (further explained in Chapter 2) to run all of the servo applications inside the photolithography machine. ASML has estimated and found by experience that this platform will not be able to cope with the amount of computation necessary to achieve a sample frequency of 100kHz with rigid body vibration compensation. In an effort to overcome this problem, research has been done to utilise Field Programmable Gate Arrays (FPGAs). Previous work (Section 2.4.1) has shown this approach has great potential, as sample frequencies of up to 154kHz have been achieved for a single servo application with non-rigid body vibration compensation deployed on an FPGA whereas it would only run at a sample frequency of at most 27kHz on a highly optimised general purpose platform (Section 2.4.2).

A big challenge that has not been addressed in the previous work is adding supervisory control that exists for the motion controller applications of ASML. This supervisory control comes from a computational unit at another network location in the computation platform. Processing of inbound supervisory control requires features normally implemented by an OS on a general purpose platform.

Implementation of the OS-like supervisory control interface on an FPGA would require a large implementation effort. Therefore, taking the big step of moving from the current general purpose processing unit to an FPGA-only processing unit is risky from a business point of view. To spread the risk, multiple steps may be done in between. In this graduation project, one of these steps will be taken: an FPGA is deployed as a Motion Controller Co-Processor (MCCP) to offload and accelerate a motion controller.

1.3 Problem Statement

The goal of this project is to get one step closer to the long term goal sample frequency of 100kHz for a future CARM motion controller, taking into account supervisory control. The research questions of the entire project are formulated as follows:

1. What parts of the servo applications should be deployed on the FPGA?
2. What kind of latency and throughput would have to be satisfied for the communication network in order to increase the sample frequency?
3. What kind of hardware/software architectures are feasible?
4. How are the best architectures implemented in the CARM framework?

Question 1 covers what parts of a servo application in general should be offloaded to the FPGA to realistically increase performance of a motion controller application. This depends on the gain in execution time when a part is executed on the FPGA, but also on the amount of data that has to be transferred from and to the FPGA.

Question 2 is more concerned with how fast the network should be in order to get to our desired performance with the options found in the first question.

Question 3 then places these ideas into an actual hardware/software architecture which will be experimentally evaluated.

Question 4 concerns the way to implement the most successful architectures into CARM. This does not include actually implementing these changes, but rather motivating them, describing the impact and proposing them for future work.

Because of the scale of the project, it is split up in two parts: a software part which is concerned with the implementation of offloading parts of a servo application on the GPP platform, while the hardware part is concerned with the design and implementation of the MCCP FPGA design and the interface it should offer to the motion controller. These two parts are carried out by two different students. This report covers the hardware part, and therefore question 4 is not covered in this document.

1.4 Solution Direction

In this project, an FPGA is deployed in conjunction with a GPP running a servo application. The GPP offloads computationally intensive parts to the FPGA in order to speed up the servo application. With this method, we expect to achieve a sample frequency of 40kHz with an IO latency of less than 10μs for a future CARM motion controller. A proof of concept of the MCCP is designed, implemented, and experimentally evaluated.

The problem of optimisation of an application on a heterogeneous platform is a well studied problem. In [3], a heuristic to find a division between hardware and software under architectural and performance constraints for an application is presented. However, the end-goal is to minimise hardware area, whereas in this project the focus is on maximisation of the sample frequency under latency constraints.

Another approach is to deploy graphics processing units (GPUs) to accelerate compute-intensive software applications. As shown in [4], GPU solutions are typically easier to implement, but are less flexible than FPGA solutions. Moreover, GPU solutions work particularly well to increase throughput when a single transformation can be applied to many input values at the same time. In servo applications, the amount of variables that undergo a single transformation during one sample period are typically small, and IO latency is more important than throughput. Hence, an FPGA solution is more suitable for this problem domain.

1.5 Document Structure

The structure of this document is as follows:

- Chapter 2: background information of the problem domain.
- Chapter 3: analysis of different aspects of the problem, resulting in an approach for motion controller offloading.
- Chapter 4: interface specification of the MCCP proof of concept
- Chapter 5: explanation of the design that implements this interface
- Chapter 6: a description of the experimental evaluation method, the results, and corresponding analysis.
- Chapter 7: conclusion and future work.

2 | Background Information

Figure 2.1 shows an example of a motion controller control loop. In this example, the forces F_x and F_y to be sent to the Wafer Stage actuator are calculated from the current sensed position (x_{sensor}, y_{sensor}) and the goal position (x_{goal}, y_{goal}). In the example, the two forces have to be decoupled because they will be applied to the same physical object.

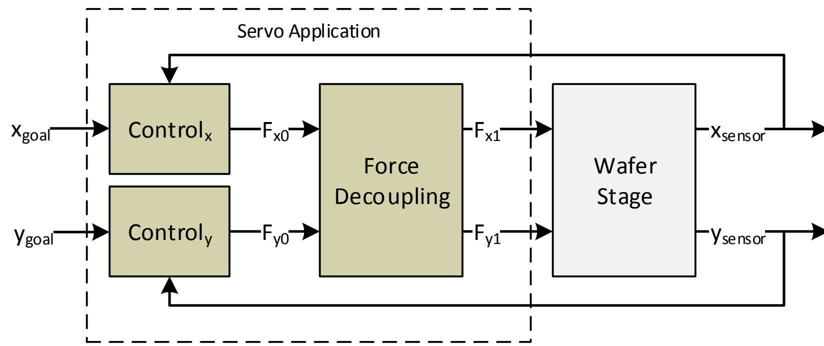


Figure 2.1: Example Wafer Stage control loop

The servo application corresponding to this control loop contains three distinct blocks of computation:

- Control_x
- Control_y
- Force Decoupling

The structure of the remainder of this chapter is as follows: the control loop example in Figure 2.1 is used to explain the current software architecture in Section 2.1. In Section 2.2, the current hardware architecture is explained. The benchmark application used for experimental evaluation is introduced in Section 2.3. Previous work done is explained in Section 2.4.

2.1 CARM Software Architecture

Control Architecture Reference Model (CARM) is a reference model devised by ASML aimed to create a well defined layered model of a system in which each layer has responsibilities at a specific abstraction level. CARM ([1]) is meant to be used in a multidisciplinary environment covering the software, electrical and mechanical disciplines.

The CARM Facilities motion controller platform is an implementation of CARM, but since it is the only implementation, it is called just CARM as well.

Servo applications such as the one shown in Figure 2.1 are implemented in CARM as Synchronous Data Flow (SDF) graphs ([5]). Synchronous data flow is a special case of data flow where the number of data samples which are produced or consumed is known a priori. Because of this property, a static schedule for the blocks in the servo application can be created at compile time. The nodes in the SDF graph of a CARM servo application are called *worker blocks*.

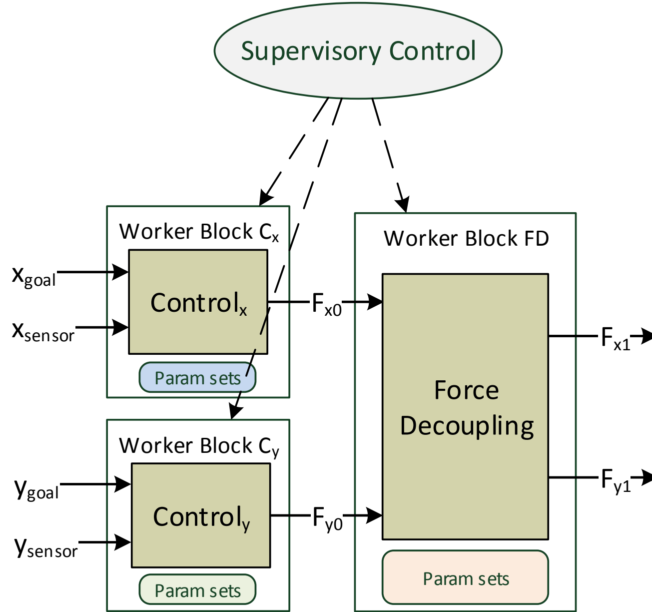


Figure 2.2: CARM version of the servo application from the control loop in Figure 2.1

Figure 2.2 shows the CARM servo application of the example control loop of Figure 2.1. Each *worker block* has its own parameter sets. A parameter set defines the values of certain variables in a block's computation function, thereby tuning the behaviour of its *worker block*. Only one of these *parameter sets* can be assigned to a block at a time, but they can be switched at run-time. E.g. a state space block, shown in Figure 2.7 is defined as follows:

$$\begin{bmatrix} \vec{x}_{k+1} \\ \vec{o}_k \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \vec{x}_k \\ \vec{i}_k \end{bmatrix}$$

Where A, B, C, and D are matrices, $\vec{i}_k$ is the input vector, $\vec{x}_k$ is the state vector, and $\vec{o}_k$ is the output vector of sample k. For this block a parameter set consists of, values for the A, B, C, and D matrices.

In addition to the regular block inputs and outputs *ports*, worker blocks have an interface for supervisory control. The supervisory control interface is used out of sync with the sample frequency of the servo application. It can be used to switch the active *parameter set* of the block or turn the block of completely. Switching the active *parameter sets* of all *worker blocks* of the servo application within the same sample period is called a *control mode switch*. Turning one or multiple blocks (a *block group*) on or off is called *state control*.

Control modes are used for different modes of execution. For example, consider the movements of the wafer stage during two different modes in the dual stage machine: `expose` mode is active during exposure of a wafer to the light, `meas` mode is active during measurement of a wafer's surface. In `expose`, movements have to be very precise, because a small error may be critical for the production process. In `meas`, making an error is less critical, so the movement speed can be increased compared to the `expose` mode. These two different modes are defined in a *control mode*. A control mode selects exactly one parameter set for each worker block in the servo application.

More formally, a parameter set is a tuple of parameter values. The type of this tuple is predefined for each worker block. Each parameter set within a block must have a distinct name for identification.

To define the relation between parameters, parameter sets, and control modes, let $\mathbb{B}$ be the set of blocks in a servo group, and $\mathcal{PT}$ be the set of all possible parameter sets for all blocks. Each block $b \in \mathbb{B}$ has parameter sets $PS_b \subseteq \mathcal{PT}$. A control mode $C : \mathbb{B} \mapsto \mathcal{PT}$ is an injective function such that

$$\forall b \in \mathbb{B} : C(b) \in PS_b$$

The number of parameter sets for each block is constant, but the values of the parameters in each parameter set can be changed at runtime. The feature of changing the values of parameters at run-time is, however, only used during calibration of the machine and is therefore left out of scope for this work.

2.1.1 Terminology Overview

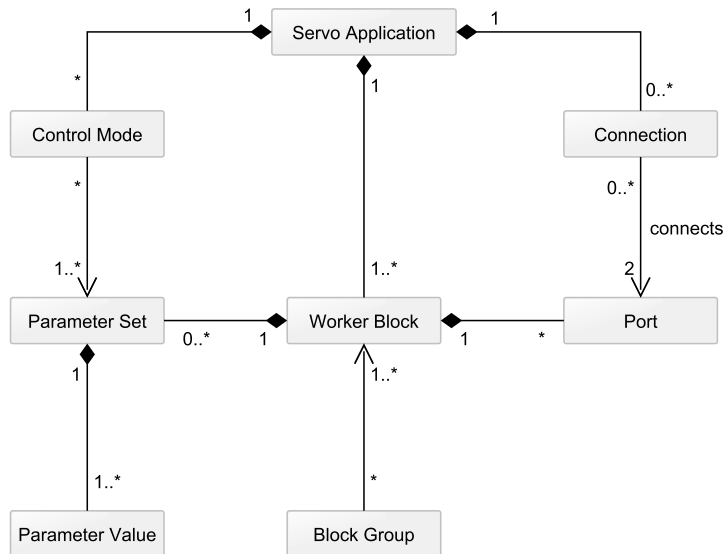


Figure 2.3: The relations of relevant high-level terms used in CARM.

Some of the more relevant high-level CARM terms are shown with their relations in Figure 2.3. The entities shown in this figure are:

- *Servo Application*: A set of *blocks* and the *connections* between them, forming an SDFG. Implements a big part of a motion controller. E.g. the long stroke/short stroke controller: Figure 2.17 and Figure 2.18.
- *Worker Block*: Also called just *Block*. A *Worker Block* can be a sensor interface, measurement system, servo controller, etc. In Figure 2.17 and Figure 2.18, examples of worker blocks are *CO_SPG_FF* and *SS_220*.
- *Parameter Set*: A set of parameter valuations that can be tuned to influence the behaviour of a block.
- *Control Mode*: Describes exactly one *Parameter Set* for each worker block in a servo group. *Control Modes* can be switched synchronously with respect to the sample period for all blocks within the *Servo Group*.
- *Block Group*: A subset of blocks of a *Servo Group*. *Block Groups* can overlap. They are used for synchronously (with relation to the sample period) controlling state behaviour (on/off) of multiple blocks at the same time.

2.1.2 Implementation

In the software implementation of the CARM motion controller platform, the supervisory control is deployed on a software platform called the Host, while the worker blocks run within a software platform called a Worker. A Host may instantiate and provide supervisory control for multiple (closely related) servo applications, using multiple Workers. An example of an instantiation of one servo application with n worker blocks and m control modes is depicted in Figure 2.4. In this figure, the arrows represent creation dependencies.

- "Application" represents an out-of-scope entity which gives the trigger to start the system.
- Host contains a Process Control Manager which instantiates a Process Control Worker on a Worker, and a Servo Application Proxy for each servo application it is supervising.
- Worker is a computation platform which runs a process control worker which can be used to run worker blocks. It has a block factory responsible for instantiating the local blocks, and a sequencer to schedule the blocks in a predefined order.

Servo application instantiation starts at the host. The host makes remote procedure calls to the block factory to create each of the servo application's blocks on a worker, and sends the schedule to the sequencer. After that, its own abstraction of the servo application, the Servo Application Proxy, is created. This abstraction contains proxies to communicate with the actual worker blocks, and the control modes of the servo group. See Section 2.1.3.1 for an MSC of this process.

The proxies are used to send the initial values of the parameter sets to the blocks, and to send new values at runtime. This communication dependency is visualised in Figure 2.5 and the order is explained in Section 2.1.3.2.

The block factory on the worker is present to instantiate blocks and connect them via a shared memory location. The sequencer enforces a schedule on the blocks by directly calling

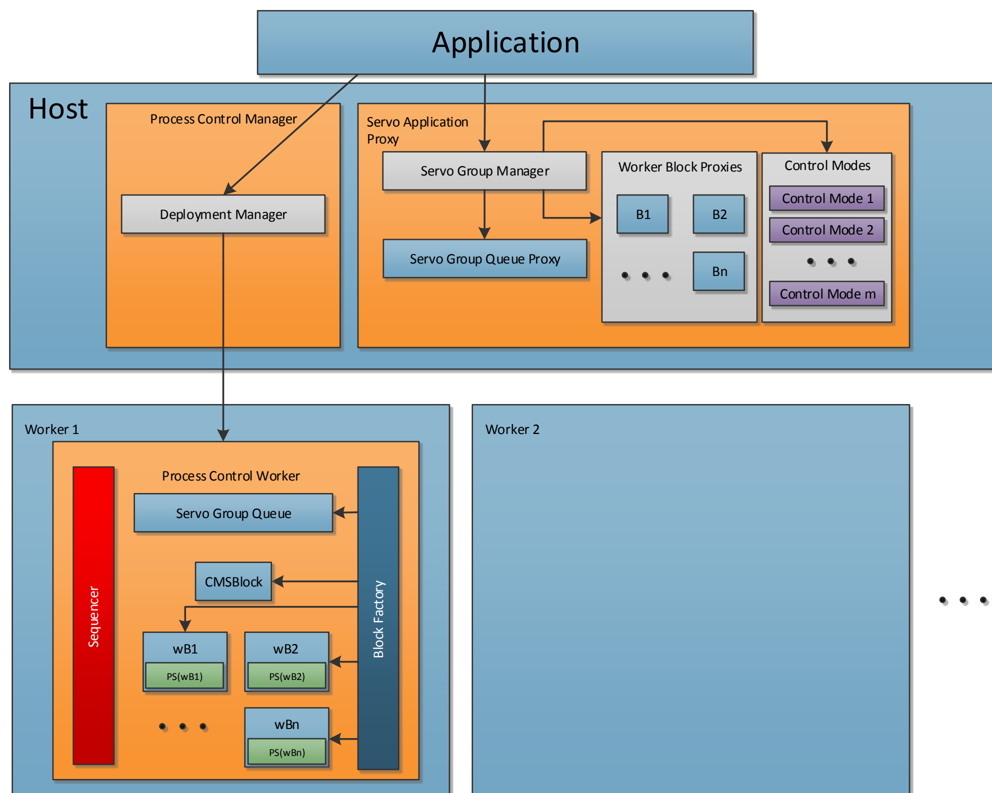


Figure 2.4: A servo group instantiation deployment diagram with arrows representing creation dependencies. PS(wBx) are all parameter sets for worker block x

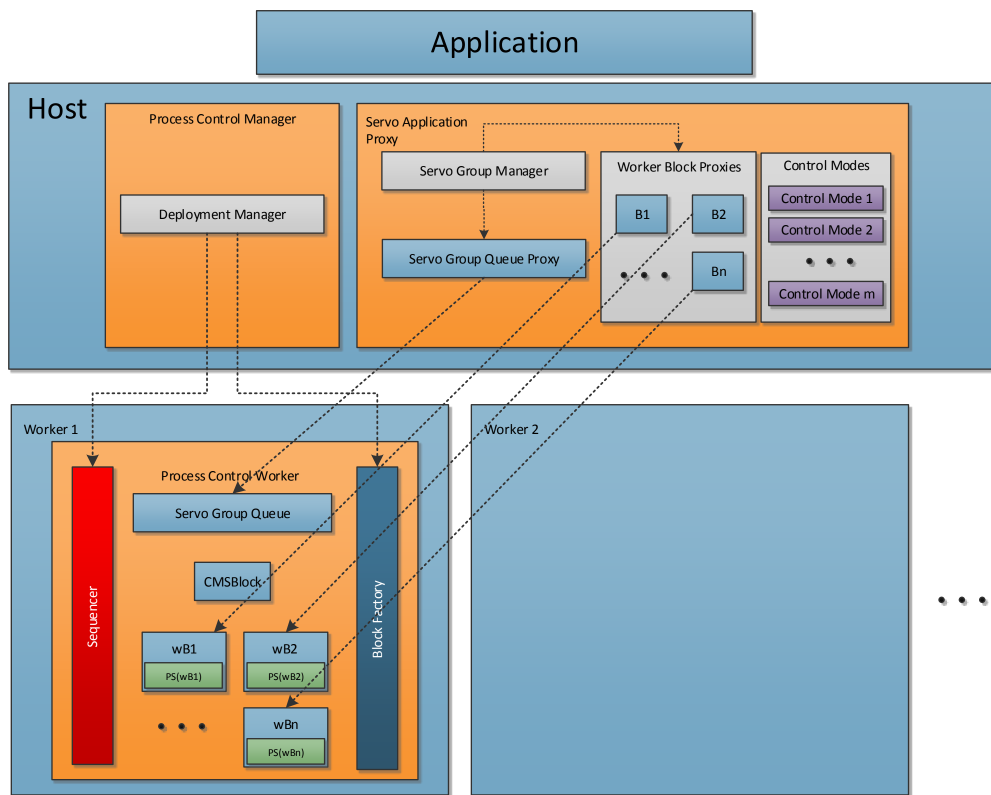


Figure 2.5: A servo group instantiation deployment diagram with arrows representing communication within and from the Host

the calculation functions (described in the next paragraph) on their instantiations. This communication is visualised in Figure 2.6. Note that in this figure, the communication between the blocks has been left out for clarity of the figure.

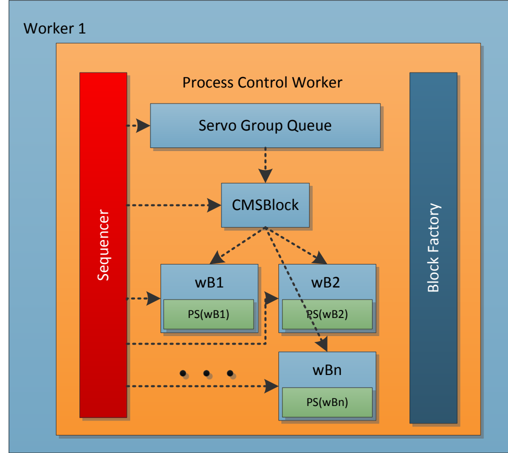


Figure 2.6: A servo group instantiation deployment diagram with arrows representing communication within an worker

A block can have three calculation functions that form the interface accessible by the sequencer. These functions are:

- `full_calc()` (`full` in short)
- `pre_calc()` (`pre` in short)
- `post_calc()` (`post` in short)

`full_calc()` is present for all blocks. This function does a full calculation to generate a (set of) output(s). To minimise the IO-delay, some blocks have `full_calc()` split up in `pre_calc()` and `post_calc()`. The composition of `pre_calc()` and `post_calc()` will yield the same result as `full_calc()`. E.g. a State-Space block shown in Figure 2.7, has a `pre` that is defined by

$$\vec{x}_k = A\vec{x}_{k-1} + B\vec{i}_{k-1}$$

$$\vec{t}_{k-1} = C\vec{x}_k$$

and a `post` that is defined by

$$\vec{o}_k = \vec{t}_{k-1} + D\vec{i}_k$$

Since `pre` only uses the inputs of the previous sample, the big matrix calculations in `pre` can be calculated after the outputs for the previous sample have been calculated. Whenever a new input arrives, $\vec{o}_k$ can be calculated with just a small matrix-vector multiplication and a vector addition. In this way, the latency of the full calculation is hidden, while the IO delay is decreased.

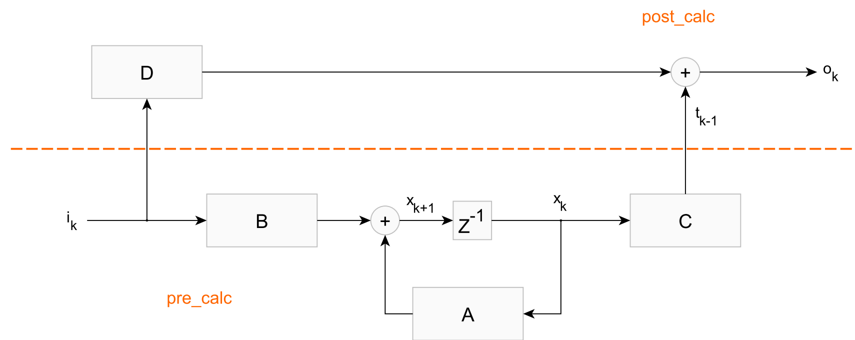


Figure 2.7: Block diagram of a State Space block

`post_calc()` should in general be as small as possible to decrease the IO delay. The division between `pre` and `post` is dependent on where in the block calculation the inputs for the next sample are used.

The sequencer's (static) schedule for each sample period is divided into two sections: time-critical and non-time-critical. For each block, a boolean can be set to define whether it is time-critical or not. The flowchart in Figure 2.8 shows how to decide in which section each function should be called. Figure 2.9 shows an example of a schedule to illustrate the the two sections. In this Figure, in addition to the sequencer's two sections, a section reserved for background operations is shown. This is used for executing supervisory control operations.

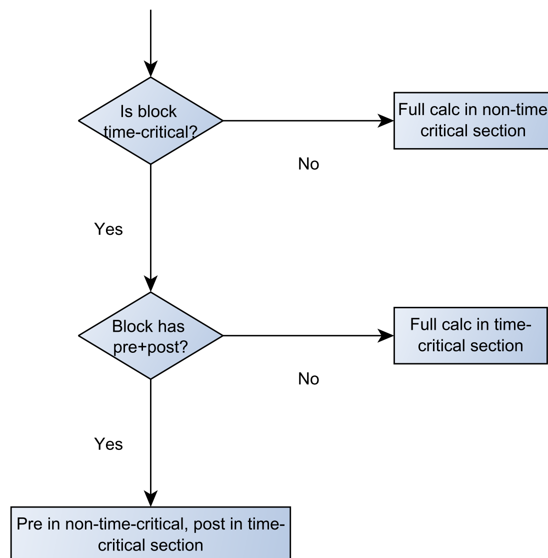


Figure 2.8: Flowchart of decision when calculation functions are called

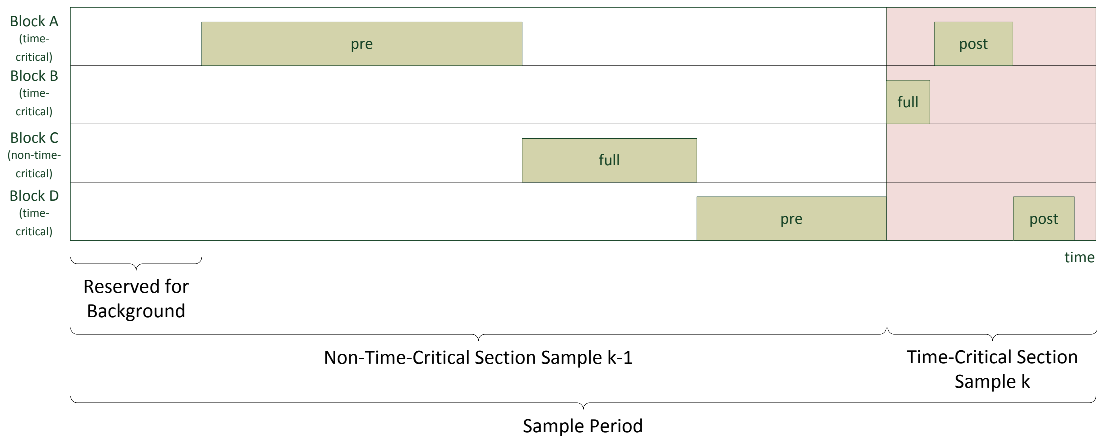


Figure 2.9: An example of a schedule of 4 different blocks. Block B is time-critical, but does not have the `pre` and `post` functions, so `full` is called in the time-critical section

2.1.3 Execution Scenarios

This section explains the following scenarios of the CARM software architecture:

- Section 2.1.3.1 explains the initialisation procedure
- Section 2.1.3.2 explains communication between the Host and Worker before and during execution of the servo application
- Section 2.1.3.3 explains how synchronous actions such as control mode switches are done

2.1.3.1 Initialisation

In Figure 2.10, the creation of a servo group is shown as a message sequence chart. The chart shows a call from the overall application to the Deployment Manager on the host. The deployment manager takes care of calling the Block Factories to create the blocks at different workers, and making the execution order of the blocks known to the Sequencer. After this is done, the Host part of the Servo Group Manager can be created. The Servo Group Manager then creates the proxies used for communication with the actual blocks.

2.1.3.2 Communication

During the initialisation phase, the proxies are used to directly communicate with the blocks via remote procedure calls. In Figure 2.11 a message sequence chart of the initialisation of the parameter sets is shown. Each parameter set is sent through the proxies directly to the block.

During execution of the servo application, the communication to the blocks happens through a different entity that acts as an operating system. This entity ensures that no race conditions occur on the main memory (for example, new parameter values are stored at the same time as it is being loaded into the cache). An example of this communication is

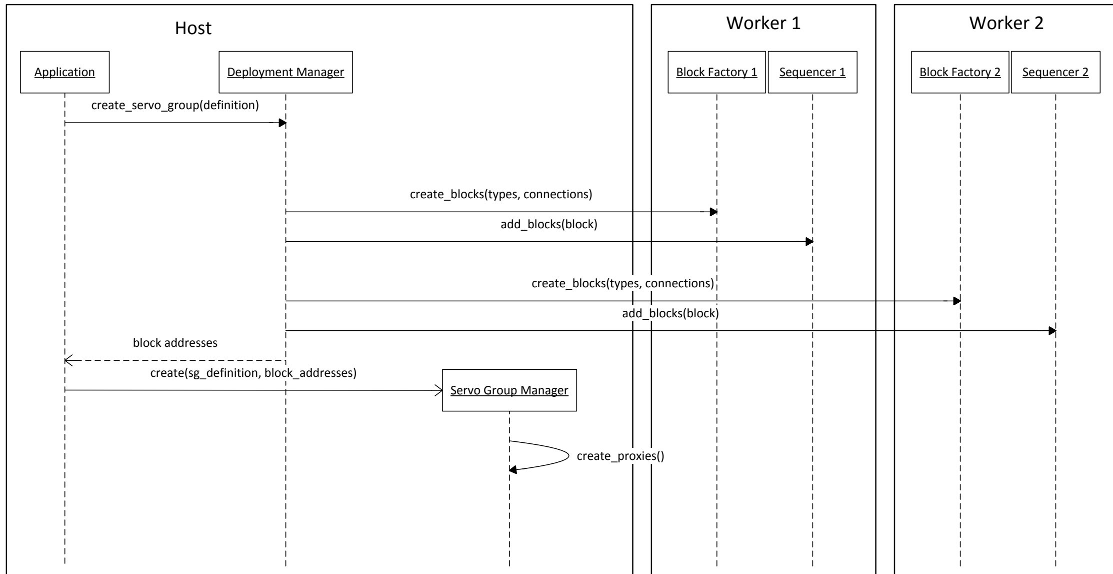


Figure 2.10: Servo group creation

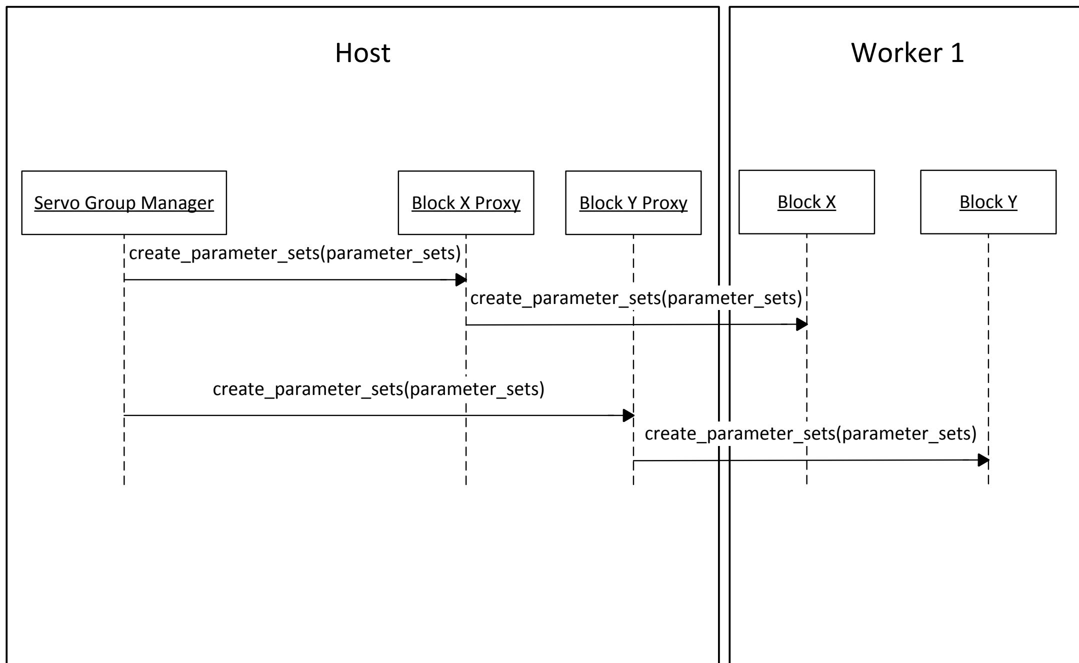


Figure 2.11: Initialisation of the parameter sets

shown in Figure 2.12. In this chart, the proxies send their messages to OS, with a destination address. The OS saves the messages for some time, and then passes them to the worker blocks when they are ready to receive messages. This happens during the section reserved for background operations of the schedule.

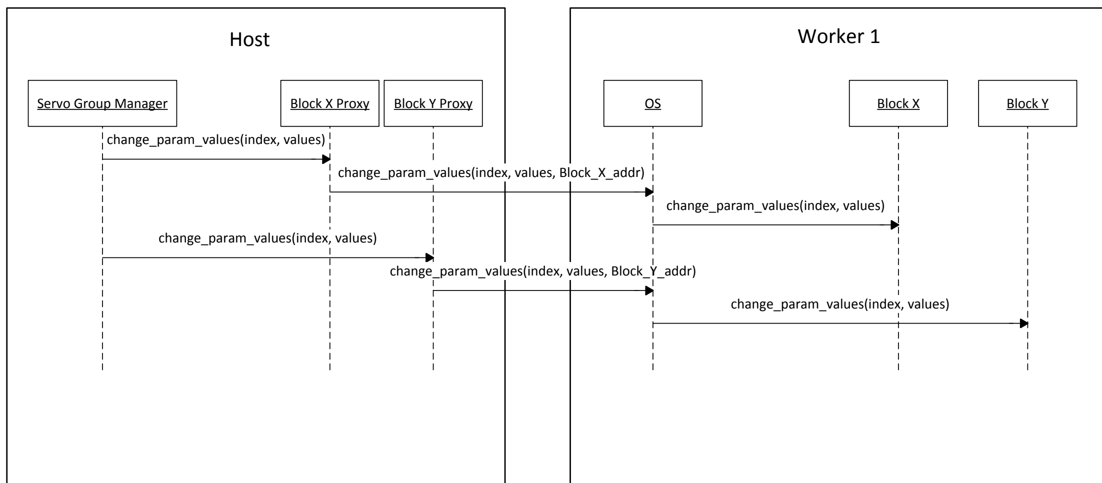


Figure 2.12: Asynchronous calls during execution of the controller

2.1.3.3 Synchronisation and Control Mode Switches

A control mode switch is the synchronous switch from one control mode to another, meaning that all worker blocks have to switch from their current parameter set to a new one at the same time before the calculation for the next sample starts. Since the parameter sets are stored in the main memory and the current parameter set is read from cache, the new parameter sets have to be preloaded into the cache for all blocks before this synchronous switch can be triggered.

To solve this synchronisation problem, a control-mode-switch block and a queue are used (Figure 2.13). The queue block is used to queue actions that take at most a predefined amount of time, such that these actions are executed in order. In case of a control mode switch, the queue passes the action to the control-mode-switch block, which in turn gives each of the targeted blocks a signal to start loading the parameter set which the block should switch to from the main memory to the local cache. Only after all blocks have loaded their parameter set, the signal is given to switch. Note that this preloading process may take up to 50 sample periods, and is done during the background operations section, so the regular data flow is not interrupted during this process.

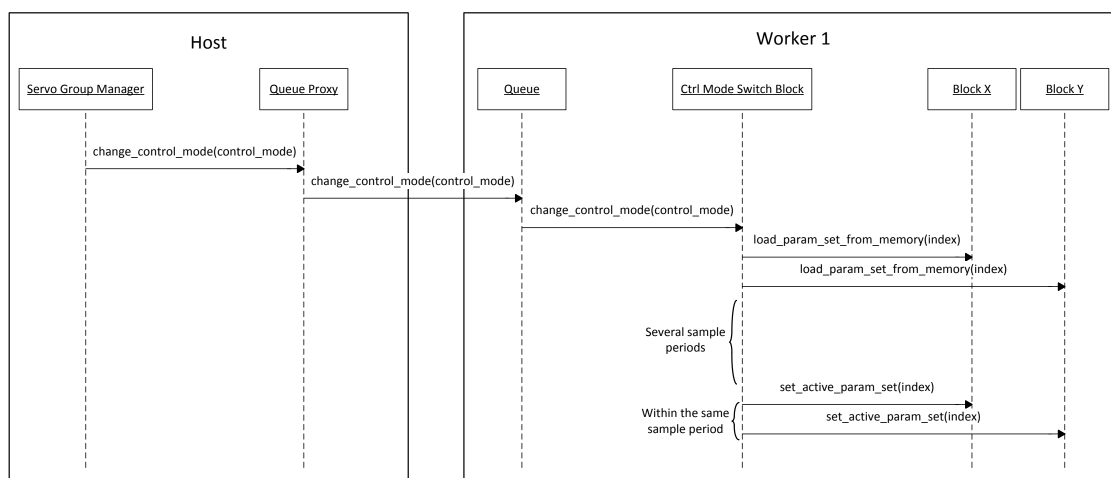


Figure 2.13: MSC of a control mode switch

2.2 CARM Hardware Architecture

In the current CARM hardware architecture, the host and each of the workers are deployed on their own computation platform called an HPPC (Section 2.2.2). The HPPCs are placed in ATCA racks (Section 2.2.1). The ATCA rack contains multiple RapidIO switches (Section 2.2.3).

2.2.1 ATCA Racks

An ATCA rack is shown in Figure 2.14. On an ATCA rack, ATCA blades are plugged, which has connectors for AMC cards. All AMC cards on a blade (which can also be the sensors inputs and actuators outputs) are connected to two RapidIO switches with a quad-lane 2.5 GBaud link (1GB/s data-rate) (Figure 2.15a). Both switches together are connected to all four of the other ATCA blades on the rack with a quad-lane 3.125 GBaud link (1.25GB/s data-rate), forming a the network as shown in Figure 2.15b. Hence, AMC cards on the same blade can be reached in one hop, and AMC cards on other blades can be reached in at least three and at most five hops, dependent in the routing scheme.

2.2.2 HPPC

High Performance Process Controllers (HPPCs) in the CARM hardware architecture are AMC cards with a (general purpose) processor on it. It can be either a PPA8548 with a Freescale MPC8548 processor, or QOR4080 with a Freescale P4080 processor.

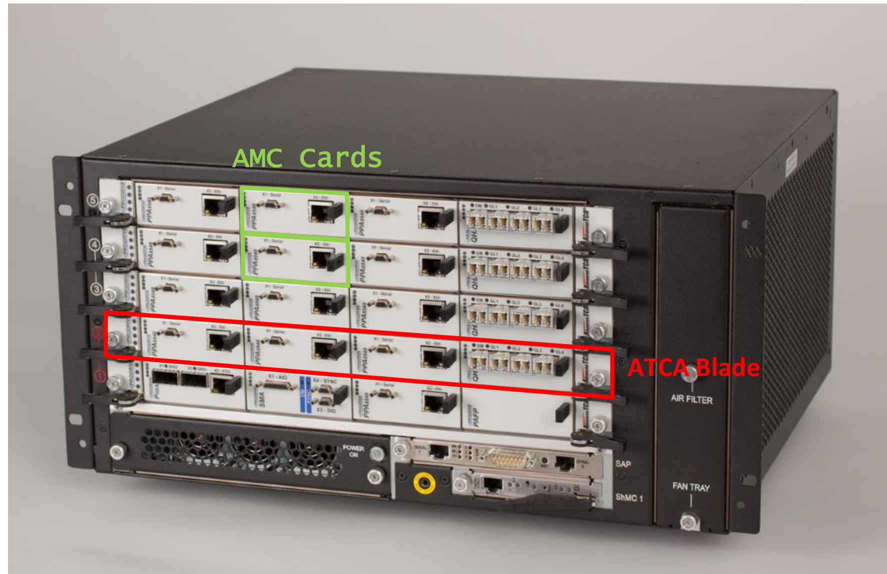
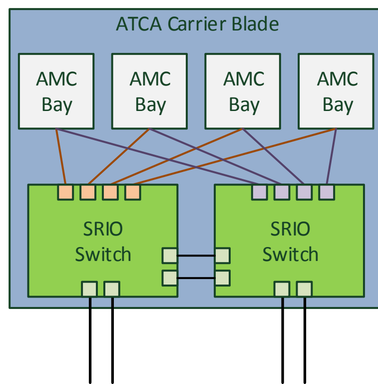
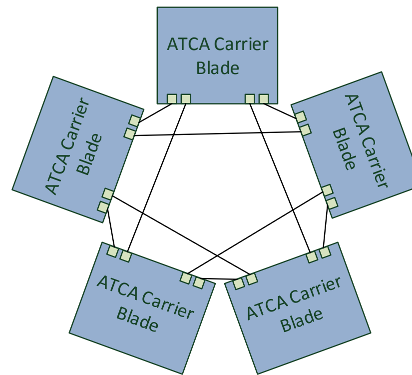


Figure 2.14: An ATCA rack showing the ATCA blades and AMC cards



(a) Physical topology of ATCA blades. The links between the AMCs and the SRIO switches are 2.5 GBaud quad-lane SRIO



(b) The network formed by all ATCA blades on an ATCA rack. The links between the blades are 3.125 GBaud quad-lane SRIO

2.2.3 RapidIO Interconnect

RapidIO is a packet switched interconnect targeted for use in environments where multiple devices must work in a tightly coupled architecture ([6]). It is specified in a 3-layer architectural hierarchy:

- Logical Specification: information necessary for the end point to process the transaction
- Transport Specification: information to transport packet from end to end in the system
- Physical Specification: information necessary to move packet between two physical devices

RapidIO has two distinct logical layer IO types:

- Message passing: packets with (a MESSAGE) or without (a DOORBELL) data are sent. For each message packet, a corresponding response packet has to be sent back to the message sender.
- Memory mapped: a piece of virtual memory is mapped to a remote device as shown in Figure 2.16.

However, it is possible to use both IO types in the same system.

Due to the unpredictability caused by the interrupts generated for arriving messages and responses, message passing is not used in the CARM motion controller platform.

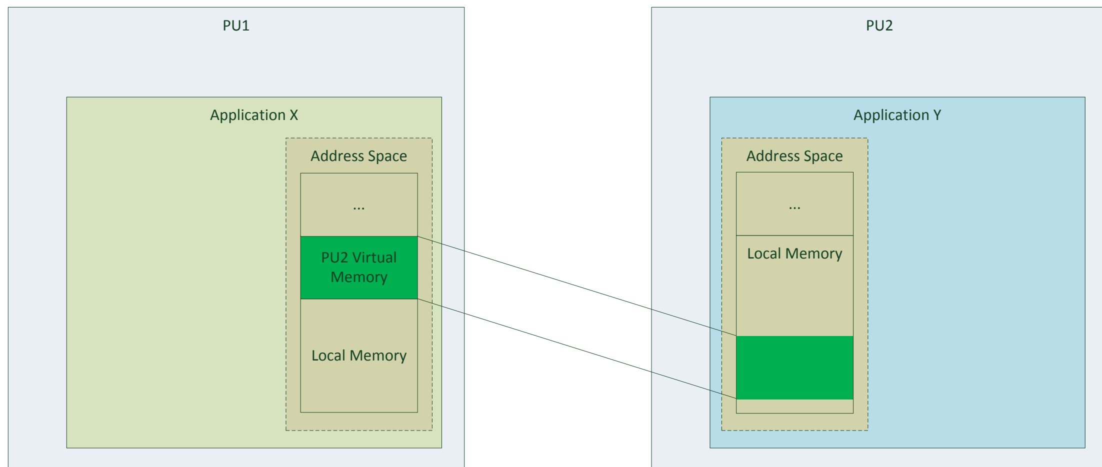


Figure 2.16: Memory Mapped Concept

Virtual memory mapped to a remote device via RapidIO is connected to a special hardware component on the HPPCs called Address Translation and Mapping Unit (ATMU). Reads and writes from the application to the virtual memory are translated to the following memory mapped RapidIO operations specified in [7]:

- NWRITE: a write to the remote memory
- NWRITE_R: a write to the remote memory with a response from the remote device on completion of the write
- SWRITE: a streaming write to the remote memory with less overhead compared to NWRITEs
- NREAD: a read on the remote memory

2.3 Benchmark Application

The benchmark application is shown in Figure 2.17 and Figure 2.18. It was created to represent a future servo application, while still being manageable for benchmarking purposes. Together, the long stroke and short stroke applications control the motion of the wafer stage or the reticle stage. The high accuracy actuator (Lorentz actuator) used for the short stroke only has a range of about 1 mm. The long stroke has a much larger range, but with only micrometer-range accuracy. The long stroke controller can move the wafer close enough such that the short stroke controller can move the wafer to the exact location.

For this graduation project, the focus lies on the SS_220 block, which is shown in Figure 2.7 and explained in more detail in section Section 2.3.1.

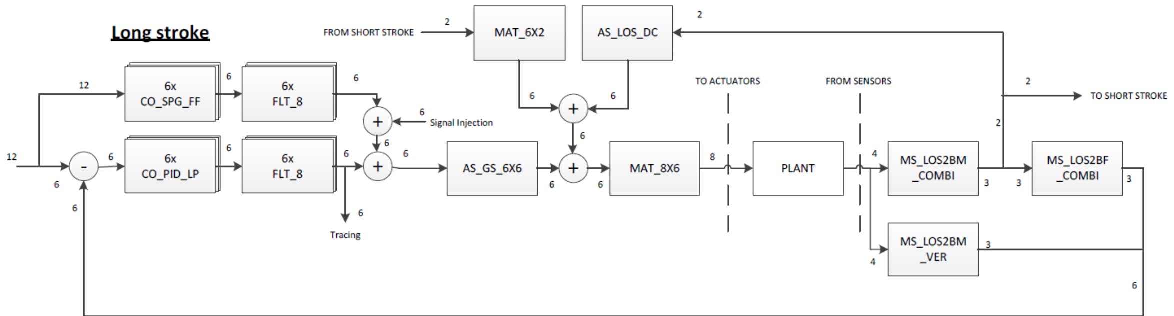


Figure 2.17: Long stroke servo group of the benchmark application. Numbers on edges represent amount of 32 bits values being consumed

2.3.1 State Space Calculation

A state space block is a MIMO representation of a linear system. A state space block with n_x internal states, input size n_i and output size n_o has matrix dimensions A:(n_x, n_x); B: (n_x, n_i); C: (n_o, n_x); D: (n_o, n_i). E.g. the SS_220 block FF in Figure 2.18 has $n_x = 220$, $n_i = n_o = 11$.

For state vector $\vec{x}_k$, the next state $(\vec{x}_{k+1})$ and the output $\vec{o}_k$ can be calculated as

$$\begin{bmatrix} \vec{x}_{k+1} \\ \vec{o}_k \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \vec{x}_k \\ \vec{i}_k \end{bmatrix}$$

where $\vec{i}_k$ is the kth input vector. From these equations, we can deduce that the output vector of the kth sample is not only dependent on the input, but also on the current state vector $\vec{x}_k$.

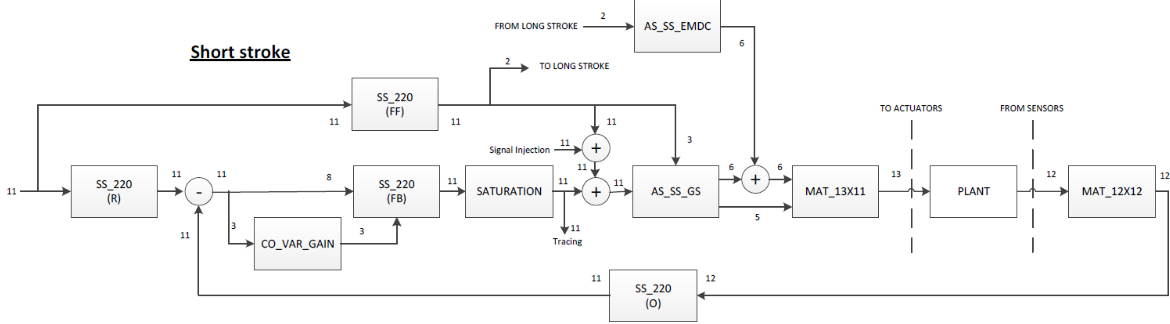


Figure 2.18: Short stroke servo group of the benchmark application. Numbers on edges represent amount of 32 bits values being consumed

2.4 Previous Work

In this section, previous work done specifically as preparation for the step to be taken in this project is explained.

2.4.1 Dataflow-based Multi-ASIP Platform Approach for Digital Control Applications

In [8], a whole servo application's SDFG has been deployed on an FPGA. Two methods have been explored, namely a direct spatial implementation of the SDF graph (see Figure 2.19), and a multi-ASIP-on-FPGA (see Figure 2.20) approach. In the multi-ASIP approach, a multi-ASIP platform is designed for the servo application. This multi-ASIP platform is then mapped onto an FPGA. This allows for high resource sharing, especially compared to the spatial approach, where each worker block gets his own resources on the FPGA (depending on the synthesis parameters). Note that the multi-ASIP platform could also be designed for a larger range of applications.

The big advantage of the multi-ASIP approach becomes apparent when, instead of deploying it onto an FPGA, it is made into an integrated circuit. It would then still be possible to change the servo application by making changes to the tasks and/or adding more tasks. Clearly, if the spatial mapping would be made into an integrated circuit, it would not be possible to maintain this flexibility.

The results of the simulations are compared to the performance of a general purpose multi-core multi-processor platform (GPP) and are shown in table 2.1. Even though the direct spatial implementation is clearly faster than the multi-ASIP-on-FPGA approach, the resource usage is significantly higher and the flexibility of this approach is much lower. This leads to the conclusion that the multi-ASIP-on-FPGA approach is a good trade-off between performance and flexibility.

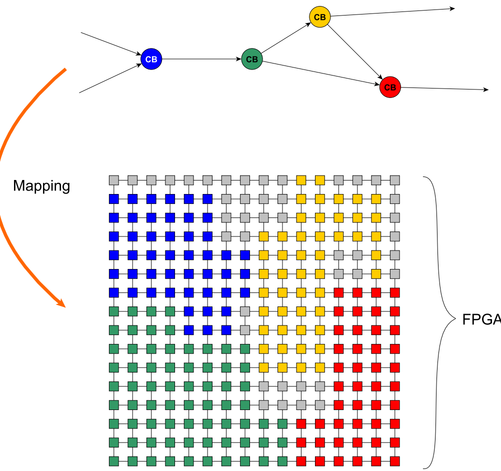
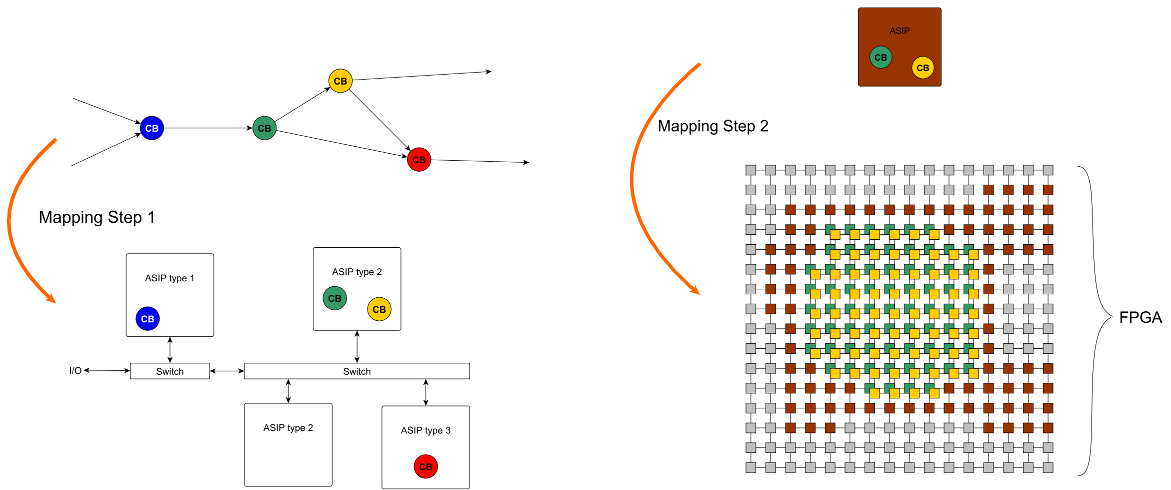


Figure 2.19: Example of a partial motion controller DFG directly mapped to a part of an FPGA.



(a) Step 1 of the ASIP-on-FPGA approach. The nodes of a DFG are mapped as tasks onto ASIPs.

(b) Step 2 of the ASIP-on-FPGA approach. ASIPs are mapped onto the FPGA. Some of the resources within the ASIP are shared between the tasks mapped onto it.

Figure 2.20: Illustrations of the multi-ASIP-on-FPGA approach.

Performance	GPP	Multi-ASIP FPGA	Spatial FPGA
I/O-delay [μs]	16	10	8.7
Sample-freq. [kHz]	27	133	154
Resource Utilisation			
Logic	-	28%	86%
Memory	-	26%	23%
DSP	-	9%	55%

Table 2.1: Results of the previous work ([8])

2.4.2 A Design-Space Exploration for High-Performance Motion Control

In [9], design-space exploration is done in order to maximise the performance of a motion controller. As a case study, a special benchmark application of the long stroke controller (LS) and short stroke controller (SS) used to represent the future generations of controllers is first benchmarked and then optimised using the Y-chart paradigm.

Table 2.2 shows the mean execution times obtained by running the application 1000 times. The most important conclusion to be drawn from this is that the state-space (SS_220) block induces most of the load by far, as it takes up 91.87% of the execution time. Hence, optimising this block would yield a significant speedup for the whole application. It is benchmarked and optimised separately on a GPP. The final optimised execution times, along with the execution times on a Vector ASIP on FPGA are shown in Table 2.3.

Job	Function	Mean Time (μs)	Load (%)
SS CTRL	SS_220	321.25	91.87
LS PreG	FLT_N	1.875	3.21
LS PreG	AS_GS_6x6	15	1.07
SS Gain	Mat_13X11	15	1.07
Other		40	2.87

Table 2.2: Initial benchmark results of the previous work in ([9])

	Pre Execution Time (μs)	Post Execution Time (μs)
GPP	51.7	0.403
ASIP-on-FPGA	7	0.8

Table 2.3: Benchmark results of the optimised SS_220 block on GPP and ASIP-on-FPGA

In the rest of the report, design-space exploration on the hardware architecture is done using these values while focussing on the state-space block. Results show that adding four vector-ASIPs to the processor to execute the SS_220 blocks does increase the performance of the motion controller dramatically to a sample frequency of 53kHz. However, the scenario is unrealistic in general, but also for our purposes, as the processor is modelled to have the vector-ASIPs on-chip. The communication overhead for block input and output, and synchronisation to an off-chip co-processor is not taken into account.

3 | Analysis and Approach

In this chapter, the CARM hardware/software architecture is analysed with respect to the following aspects:

- Block Calculation (Section 3.1)
- Block Offloading (Section 3.2)
- Control Mode Switches (Section 3.3)
- State Control (Section 3.4)
- Hardware Architecture (Section 3.5)

The analysis results in the required functionality for the Motion Controller Co-Processor (MCCP) in the existing motion controller framework. The sample frequency of the solution is predicted in Section 3.6. In Section 3.7, the expected usage of the MCCP is explained.

3.1 Block Calculation

The basic functionality of the MCCP is that it has to be able to execute calculation functions of worker blocks.

[8] proposed an approach with a multi-ASIP platform for its high flexibility compared to a direct FPGA implementation of the servo application (see Section 2.4.1): a single static multi-ASIP platform can be programmed to execute a range of different servo applications.

Using the same type of multi-ASIP platform in the MCCP has the following advantages:

- The MCCP is decoupled from the exact (parts of) blocks it will calculate.
- The multi-ASIP platform has been tested extensively, and is known to perform well for worker block calculations. This saves time in the implementation and design of the proof of concept.

Therefore, the same type of multi-ASIP platform will be used as calculating core of the MCCP. In order to be able to execute the state-space blocks with the highest amount of parallelism, the ASIP platform in the MCCP will contain one Vector ASIP (VPE) for each SS_220 block in the servo application. To connect these VPEs to the rest of the design, one External Interface, and a Switch to connect them all together will be used. The resulting platform is shown in Figure 3.1.

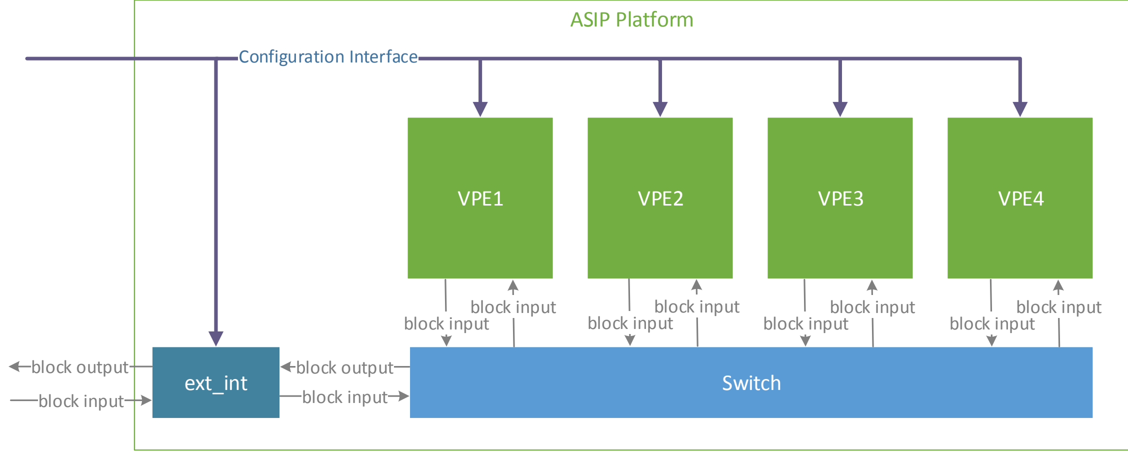


Figure 3.1: ASIP platform configured for computing four SS_220 blocks in parallel

3.2 Slack Estimation

As described in section 2.4.2, the SS_220 block takes up over 90% of the execution time of the benchmark application. Using a vector ASIP, a speedup of over 10 for the complete application can be achieved. A more detailed look shows that this gain comes mainly from the `pre` part of the execution, whereas the `post` part takes 2 times (0.4μs) longer. In this section, the gain in calculation time when offloading a certain part of the SS_220 block to the MCCP is compared to the communication overhead it introduces.

There are two viable options to improve performance of the motion controller: offload `pre` only and offload both `pre` and `post`. Note that offloading only `post` is not viable, since it would take longer on FPGA, so no speedup could be achieved. Calculations can be done to assess the profit that will be achieved by both options. Recall from section 2.4.2 that the execution time of `pre` of the optimised SS_220 block is $\tau_{pre_GPP} = 51.8\mu$s on the GPP, whereas the execution time on a vector ASIP was $\tau_{pre_ASIP} = 7\mu$s. t_{slack_pre} is the time freed up in the non-time-critical section of a sample period and can be calculated by

$$t_{slack_pre} = \tau_{pre_GPP} - \tau_{pre_ASIP} = 51.8 - 7 = 44.7\mu\text{s}$$

The freed up time for the time-critical section, in case `post` is offloaded as well is

$$t_{slack_post} = \tau_{post_GPP} - \tau_{post_ASIP} = 0.4 - 0.8 = -0.4\mu\text{s}$$

This means that there is $0.4\mu s$ less time in the time-critical section.

If only `pre` is offloaded, then this means that the 44.7μs freed up in the non-time-critical section can be used to communicate the inputs and outputs of the `pre` function. If both `pre` and `post` are offloaded, then no new inputs have to be sent to the FPGA in the non-time-critical section, since it uses the input vector of the previous `post` calculation. However, in the time-critical section, both the input and output of the `post` function have to be communicated.

This would directly increase the IO latency of the system, whereas it would not be increased if the communication overhead is hidden in the non-time-critical section.

To get an estimate of how much data can be sent in the non-time-critical section to break even, assume, for a link between two AMC cards, a latency of $2\mu s$ and an *available bandwidth* of $300MB/s = 300B/\mu s$. Then

$$(t_{slack_pre} - 2 * 2\mu s) * 300B/\mu s = (44.7\mu s - 2 * 2\mu s) * 300B/\mu s \geq data_{input} + data_{output}$$

So the maximum amount of data that can be transferred within this time in the non-time-critical section is 12210B or 3052 32-bit values.

It is important to keep in mind that the size of the output from the `pre` function could be significantly higher than that of the `post` function, possibly cancelling out any profit obtained by executing on the FPGA. Inspection of the `pre` and `post` yields that the only result necessary from `pre` to compute the `post` is one vector of the size of the output. The biggest input size present in the benchmark application is 12, and the biggest output size is 11. The time saved during the non-time-critical section when offloading only `pre` can be estimated as:

$$\frac{(3052 - data_{input} - data_{output}) * 4B}{300B/\mu s} = \frac{(3052 - 11 - 12) * 4B}{300B/\mu s} = 40.3\mu s$$

When offloading both `pre` and `post`, in the non-time-critical section, the amount of time that can be saved can be calculated as:

$$\frac{3052 * 4B}{300B/\mu s} = \frac{3052 * 4B}{300B/\mu s} = 40.7\mu s$$

But in the time-critical section:

$$|t_{slack_post}| + 2 * 2\mu s + \frac{data_{input} + data_{output}}{300B/\mu s} = 0.4 + 2 * 2\mu s + \frac{(12 + 11 * 4)}{300B/\mu s} = 4.7\mu s$$

is lost, which directly influences the IO latency.

Under these assumptions, we can conclude that when offloading only `pre`, each sample $40.3\mu s$ can be saved. Conversely, offloading post as well, $40.7\mu s$ can be saved during the non-time-critical portion, but during the time-critical portion, $4.7\mu s$ of overhead time would be added. The latter overhead directly affects the IO latency of the servo group by a significant amount.

Another advantage of offloading only `pre` of blocks is that inputs of all blocks are known after the time-critical section has finished, since it only uses inputs of the previous sample. The parallelism of the FPGA can be fully exploited by sending inputs for all offloaded tasks at the same time.

3.3 Control Mode Switches

In order to facilitate control mode switch functionality, the MCCP has to be able to switch parameter sets for offloaded blocks synchronously with the other blocks in the servo application. The first issue is that before the switch, the new parameter sets of all blocks have to be

present in the MCCP. A double buffering approach similar to the one on the HPPC-Worker is used. Figure 3.2a shows the architecture of the VPE with the double-buffered parameter sets.

Parameter sets for the SS_220 `pre` are the matrices A, B, and C. In the benchmark application, the SS_220 block O has the largest parameter set, namely:

$$220 * 220 + 12 * 220 + 11 * 220 = 53460 \text{ 32-bit values}$$

The other three blocks have parameter set size

$$220 * 220 + 11 * 220 + 11 * 220 = 53240 \text{ 32-bit values}$$

Storing all parameter sets for all offloaded blocks in the FPGA RAM would yield bad scalability properties, since the RAM resources on FPGAs are very limited. Two options remain to be explored:

- Sending new parameter sets at run-time.
- Storing all parameter sets on an external memory.

The amount of data to be transferred for a control mode switch is

$$53460 + 53240 * 3 = 266420 \text{ 32-bit values} = 0.85\text{MB}$$

The number of samples available to do parameter set pre-loading is 50, so at the goal sample frequency of 40 kHz, a bandwidth of

$$0.85\text{MB} * \frac{40000}{50} = 682 \text{ MB/s}$$

is used. Since the datarate in the ATCA blades is at most `1GB/s` (Section 2.2.1), loading the parameter sets in this manner would use over 68% of the available bandwidth during 50 sample periods. Therefore, it would interfere with the other traffic, and scalability issues arise when increasing the sample frequency beyond 40 kHz. Hence, the only viable option left is to store all parameter sets in an external memory of the MCCP at initialisation. This means that:

- When pre-loading the parameter sets for the control mode switch on the Worker, the HPPC has to send a message to the MCCP to load the parameter sets from the external memory into the FPGA RAM.
- To ensure that the parameter sets have been loaded into the FPGA RAM before doing the synchronous switch, a flag has to be sent back to the HPPC after a load has finished.

The second issue is that the MCCP, and thus the ASIP platform, should do the switch synchronously. Assume that the new parameter sets are already pre-loaded into the FPGA RAM. The active parameter sets have to be switched from the old ones to the new ones. For this, four approaches illustrated in Figure 3.2 are explored:

1. Every offloaded block has a mirrored task, which uses the same state, but a different parameter set (Figure 3.2a): to switch, the HPPC has to target the other task after the parameter set has been loaded into its parameter memory.
2. Change the memory address in the task instructions: to switch, all instructions in the instruction memory with a reference to the active parameter set have to be overwritten.
3. Use relative addressing in the instructions with a reference to the active parameter set and an offset register which points to the address of the active parameter set (Figure 3.2c): to switch, the offset register has to be overwritten.
4. Highest bit address space flip (Figure 3.2d). The memory is divided into two halves: to switch, one single highest bit should be flipped.

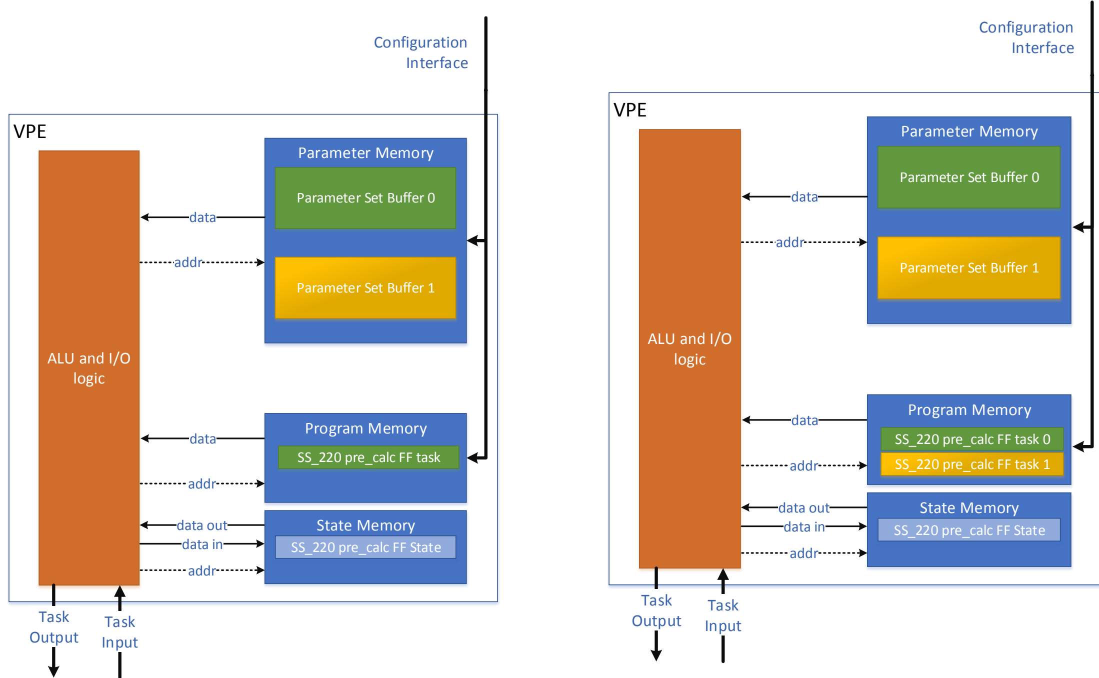
The following table shows the pros and cons of each of the approaches

	Pro	Con
1	No additional communication overhead for a switch No changes to the ASIP platform required	More ASIP instruction memory required If blocks have input/output dependencies to other blocks in the ASIP platform, these other blocks must use the parameter set corresponding to the new control mode after the switch as well [1]
2	ASIP instruction memory size unchanged No changes in the ASIP platform required	Adds communication overhead
3	Task instruction memory size unchanged Fast switching; only one word has to be sent per task	Need one extra clock cycle for computing address for every computation with a parameter set reference [2] Requires implementation effort; the ASIP platform does not currently support it Needs synchronisation; offset register has to be changed before new task inputs for the new control mode can be received
4	Fast Switching; only one word has to be sent per task	Not currently implemented in the ASIP platform Needs synchronisation; address space flip has to be registered before task inputs for the new control mode can be received Sub-optimal use of parameter memory; constant values would have to be duplicated in both parts of the memory

Recall that `pre_calc` by definition has no output dependencies to other blocks. In Section 3.2, offloading `pre_calc` was found to be the best option for gaining performance when using an FPGA co-processor. Therefore, approach 1 only has one con, namely the extra ASIP memory required. Since the size of instruction memory is insignificant compared to the size of the parameter memory for the state-space block, approach 1 is found to have the least downsides.

[1] This could be solved by either performing a parameter switch for all blocks in an internally dependent set, or implementing a configurable destination for task outputs.

[2] This latency could be hidden during the actual execution, but it changes the timing model of the ASIP, which means that the program code has to be changed.



(a) VPE with double buffered parameter sets

(b) Approach 1: VPE with shared state (colours show task to parameter set correspondence)

(c) Approach 3: VPE with parameter offset register (colour shows current offset value)

(d) Approach 4: VPE with highest address bit flip mechanism (colour shows current bit = 0)

Figure 3.2: VPE architecture with different parameter switch approaches

3.4 State Control

When a block is turned off by state control, it can either behave as a pass-through, or output a constant value irrespective of the input. Consider the case where a subgraph of the servo application shown in Figure 3.3 would be offloaded to the MCCP. Now inputs are only sent from the HPPC to block A, and outputs from block C are received. When block B is turned off, the new behaviour of the block has to be registered in the MCCP. I.e. it has to switch to a task that does not do the calculate but instead passes input to output, or constant values on the output. This switch also has to be processed synchronously with the sample frequency of the motion controller: it has to be guaranteed that the task has switched before any new inputs can be handled.

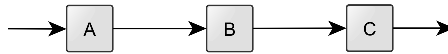


Figure 3.3: Servo application subgraph example

This issue only occurs when a dependency is present between the blocks offloaded to the MCCP. Following the same reasoning as in Section 3.3, when offloading only `pre_calc` of blocks, no dependencies to other blocks are present. `post_calc` which is still executed on the HPPC-Worker calculates the actual output of the block. Hence, state control need not be considered for the MCCP, since it can be solved at the HPPC-Worker.

3.5 Hardware Architecture

The choice of the hardware architecture of the solution and the corresponding test setup for the experimental evaluation of this project is based on several factors:

- Size of the FPGA (amount of logical units, BRAMs, etc)
- Test setup implementation effort
- CARM implementation feasibility
- Communication latency and bandwidth

The size of the FPGA covers the amount of resources (logic and memory) present on the FPGA. For the test setup, this should be more than enough, such that development of the proof of concept will not be blocked on this. In an actual real-life deployment, the size of the FPGA could be custom selected based on resource usage values obtained during development of the proof of concept.

The test setup implementation effort is the effort to get the test setup up and running. This should be as low as possible, so that the chance of obtaining useful results is increased.

CARM implementation feasibility is the degree of which the solution architecture can be implemented in the current hardware architecture. It is not desirable for the solution to

demand a complete restructuring of the current architecture, unless it is absolutely necessary in order to achieve the target sample frequency.

Communication latency between the GPP and FPGA should be low enough, while the bandwidth should be high enough such that the gain in execution time of offloaded blocks outweighs the communication penalty for offloading blocks.

3.5.1 Solution Options

To increase the feasibility of implementation in CARM, and since Section 3.2 has shown that the bandwidth is high enough, only architectures resulting from placing an AMC board with an FPGA on it in one of the ATCA blades are considered. Two options for the card are possible:

1. pure FPGA board
2. SoC with some GP processor and FPGA on it
    (a) Worker on the SoC GP processor
    (b) Worker on HPPC, SoC GP processor is part of the co-processor

The main advantage of option 2 and the reason it is considered for this project is the short latency between the GPP and the FPGA due to direct, private, communication through a shared cache.

There are roughly two ways to deploy the second option: (a) use the GP processor as a full worker, or (b) use it to take over some general purpose computation from the FPGA (like loading the parameter sets to the shared cache) and keep the actual worker on the HPPC. For both of these ways, a large amount of optimisations and trade-offs could be made in order to utilise the SoC features fully. No relevant previous work is done on the topic of optimising a CARM motion controller on such an FPGA+GPP SoC. To perform this complete design-space exploration in this project would be too big a task to take on. Additionally, Section 3.2 has shown that the latency caused by a RapidIO switch is small enough to get a speedup when offloading `pre` of blocks.

In conclusion, the second option introduces too many variables to consider, so to reduce the risk of not getting any significant results from this work, option 1 is chosen. It is expected that this work will give a better insight in whether the FPGA+GPP SoC will indeed further increase the performance of a motion controller, so that is left for future work.

3.6 Sample Frequency Feasibility Analysis

In this section, an analysis is done to estimate how many blocks can be offloaded to the MCCP before the network becomes the bottleneck. To perform this analysis, the following assumptions are made:

- HPPC computation time is ignored, and the sample has no background period.

- The MCCP is able to compute all of the offloaded blocks in parallel.
- The HPPC and MCCP can utilise a complete 2.5GBaud SRIO link, either quad-lane ($dr_4 = 1.00\texttt{GB/s}$) or single-lane ($dr_1 = 0.25\texttt{GB/s}$) with a latency of $l = 1.5\mu\texttt{s}$.
- Loading of parameter sets for all offloaded block from the external memory into the ASIP platform can be done within the prescribed HPPC parameter set pre-loading time (50 sample periods), so it does not block the sample frequency.

The communication overhead is dependent on the amount of data that has to be communicated, the latency, and the bandwidth of the channel. A general formula for the communication overhead for n blocks with $\#in$ bytes of input and $\#out$ bytes of output when values are sent back to back is:

$$IO = \frac{\#in + \#out}{dr} * n + 2 * l = \frac{\#in + \#out}{dr} * n + 2 * 1.5\mu\texttt{s}$$

The maximum achieved sample frequency f, when not taking control mode switching overhead into account can then be calculated as

$$f = \frac{1}{IO + c_{block}}$$

where c_{block} is the execution time of the blocks.

The information that is needed for a parameter set load command when the MCCP has no knowledge of the size of parameter sets for each block is

- Address in the external memory where the parameter set is stored.
- Address in the ASIP platform configuration memory space where the parameter should be loaded to.
- Number of values to load

Assume that 4 bytes are used for each of these values. A complete parameter set load command (`plc`) then consists of 12 bytes. A simple synchronisation mechanism of parameter set load commands could be sending a flag from the MCCP to the HPPC whenever a load has been done. These flags are sent uniformly spread over the parameter set pre-loading time (50 sample periods) and have a size of less than 4 bytes. When a flag arrives at the HPPC, it is written into its inbound RIO memory, which can be checked right before executing the switch to the new control mode. This feedback mechanism of only one 32-bit value does not cause any notable overhead, and is therefore ignored in the rest of the calculations. Hence, the communication overhead for n blocks for a sample with a parameter set load instruction IO_{CMS} can be calculated as

$$IO_{CMS} = \frac{\#in + \#plc + \#out}{dr} * n + 2 * l$$

and corresponding sample frequency f_{CMS}

$$f_{CMS} = \frac{1}{IO_{CMS} + c_{block}}$$

For an SS_220 `pre_calc` block, the maximum number of inputs and outputs found in the benchmark application is $11 * 4 + 12 * 4 = 92B$, and execution time $c_{block} = 7\mu\texttt{s} + 3\mu\texttt{s}$ (computation time from Section 2.4.2 with some additional internal MCCP latency). Figure 3.4 shows for $0 < n \leq 12$ when offloaded blocks are SS_220 `pre_calc` computations:

- f_1: single-lane SRIO communication channel, no control mode switch overhead
- f_4: quad-lane SRIO communication channel, no control mode switch overhead
- f_{CMS1}: single-lane SRIO communication channel, added control mode switch overhead for every sample
- f_{CMS4}: quad-lane SRIO communication channel, added control mode switch overhead for every sample

It shows that if all four blocks are offloaded, a maximum sample frequency of 68kHz could be achieved for a single-lane link. The bandwidth difference for single-lane compared to quad-lane has a bigger impact as the number of blocks increases. At the same time, the impact of the parameter load messages is very minimal, even for larger numbers of offloaded blocks.

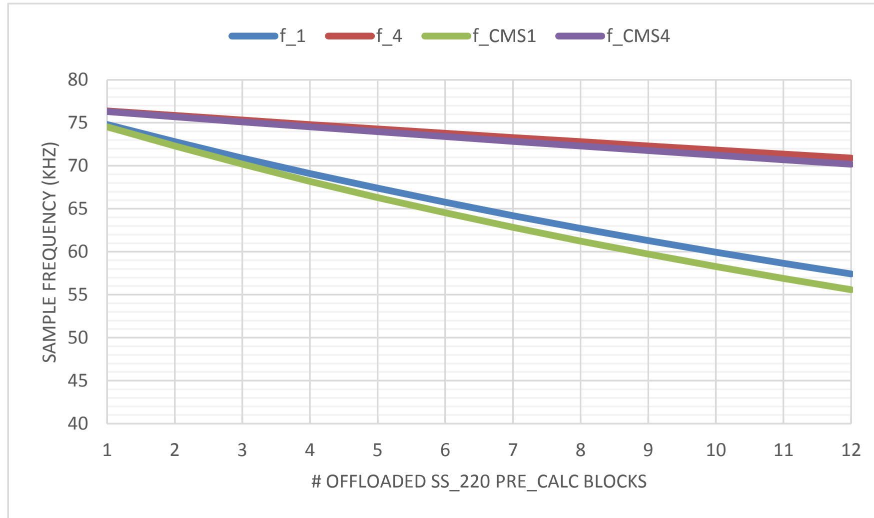


Figure 3.4: Sample frequency for $0 < n \leq 12$, $c_{block} = 10\mu\texttt{s}$, and $\#in + \#out = 92\texttt{B}$

This analysis shows that the MCCP will be able to run in a motion controller running at at most 68kHz for a single-lane SRIO communication channel or even 75kHz for a quad-lane SRIO communication channel when offloading four SS_220 blocks.

Realistically, whether all blocks could be computed in parallel on the MCCP and whether all parameter sets could be loaded within 50 samples heavily depends on the FPGA resources and peripherals, but also the design itself. Furthermore, reaching this sample frequency would require a sample schedule on the HPPC in which all non-offloaded blocks can be computed

and supervisory control from the Host handled during the time that the MCCP is computing the offloaded blocks, and no execution time in the time-critical section.

In previous work [9], explained in Section 2.4.2, a sample frequency of 53kHz was obtained for a multi-core processor with 4 vector ASIPs on-chip by optimising the sample schedule, essentially the inverse of the analysis presented above. The *minimum requirement* to be able to reach the goal of 40kHz is that the added overhead for offloading has to be less than

$$\frac{1}{40\text{kHz}} - \frac{1}{53\text{kHz}} = 6\mu\text{s}$$

This seems feasible indeed, since communication overhead on a quad-lane 2.5 GBaud SRIO link

$$IO_{CMS4} = \frac{92\text{B} + 3\text{B}}{1.00\text{GB/s}} * 4 + 2 * 1.5\mu\text{s} = 4.7\mu\text{s}$$

Leaving $6\mu\text{s} - 4.7\mu\text{s} = 1.3\mu\text{s}$ for offloading-related calculations on the HPPC. Hence, considering both of these results, a sample frequency of 40kHz when offloading all SS_220 `pre_calc` blocks in the benchmark application is feasible.

3.7 Functional Overview

Following from the analysis, the functionality and expected usage of the MCCP can be deduced.

- During initialisation, following from Section 3.1 and Section 3.3, the ASIP platform configuration (offloaded blocks program code) and parameter sets of all offloaded blocks have to be sent to the MCCP.
- Only the `pre_calc` function of blocks are offloaded, and each offloaded `pre_calc` has two tasks on the MCCP: an active task to which the inputs are currently being sent, and an additional task to which the next parameter set can be loaded.
- At each sample at the start of the non-time-critical period, according to Section 3.1, Section 3.2, and Section 3.3, block inputs for all offloaded `pre_calc` blocks have to be sent to the currently active task of each `pre_calc`.
- The non-time-critical period ends when all block outputs have been received from the MCCP and all non-offloaded blocks with calculation functions in the non-time-critical period have been done.
- When parameter sets have to be preloaded, following from Section 3.3, parameter load commands have to be sent for each offloaded `pre_calc` block to the task that is currently not active.
- When the control mode has to be switched, the non-active task is made the active task and vice-versa.

Figure 3.5 shows the initialisation where the HPPC worker sends the initialisation data, which need not be the case in the real platform. In the scenario, all four of the SS_220 `pre_calc` blocks FF, R, FB, and O are offloaded to the MCCP. Each block X has two tasks

X_0 and X_1. The scenario shows that after initialisation of the MCCP, an initial parameter preload command arrives from the host to switch to a control mode before the servo application starts executing. The initial parameter sets are loaded for tasks 0.

Figure 3.6 shows the run-time scenario where a pre-load command arrives. Parameter load commands for the offloaded blocks are sent for blocks FF_1, R_1, and FB_1. Block O uses the same parameter set after the control mode switch in this scenario, so no load has to be sent. During the 50 sample periods after the preload has started, load done messages can be sent from the MCCP. Before the control mode switch occurs, a check is done to verify that all parameter load commands have been responded to with load done messages. After the control mode switch, the inputs for the blocks are sent to the tasks where the parameter sets have just been loaded, which is the same for block O.

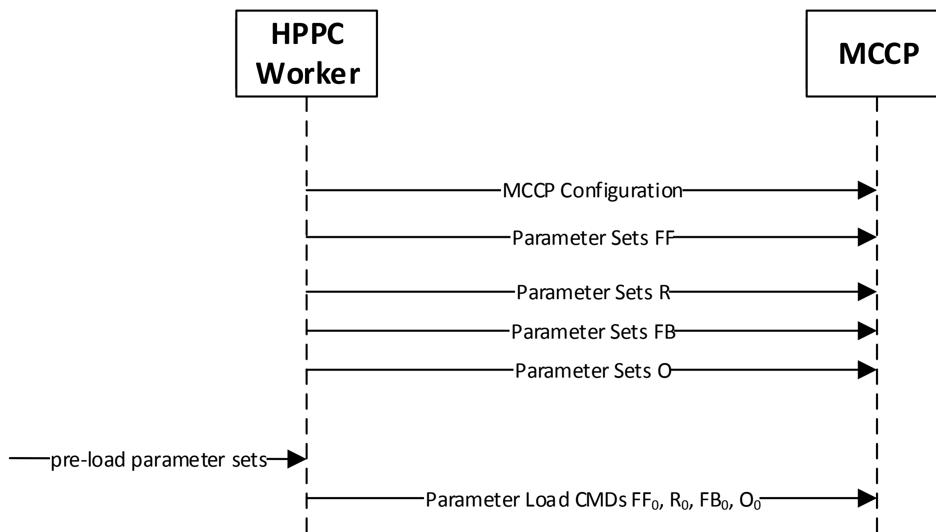


Figure 3.5: MSC of initialisation of a worker with MCCP

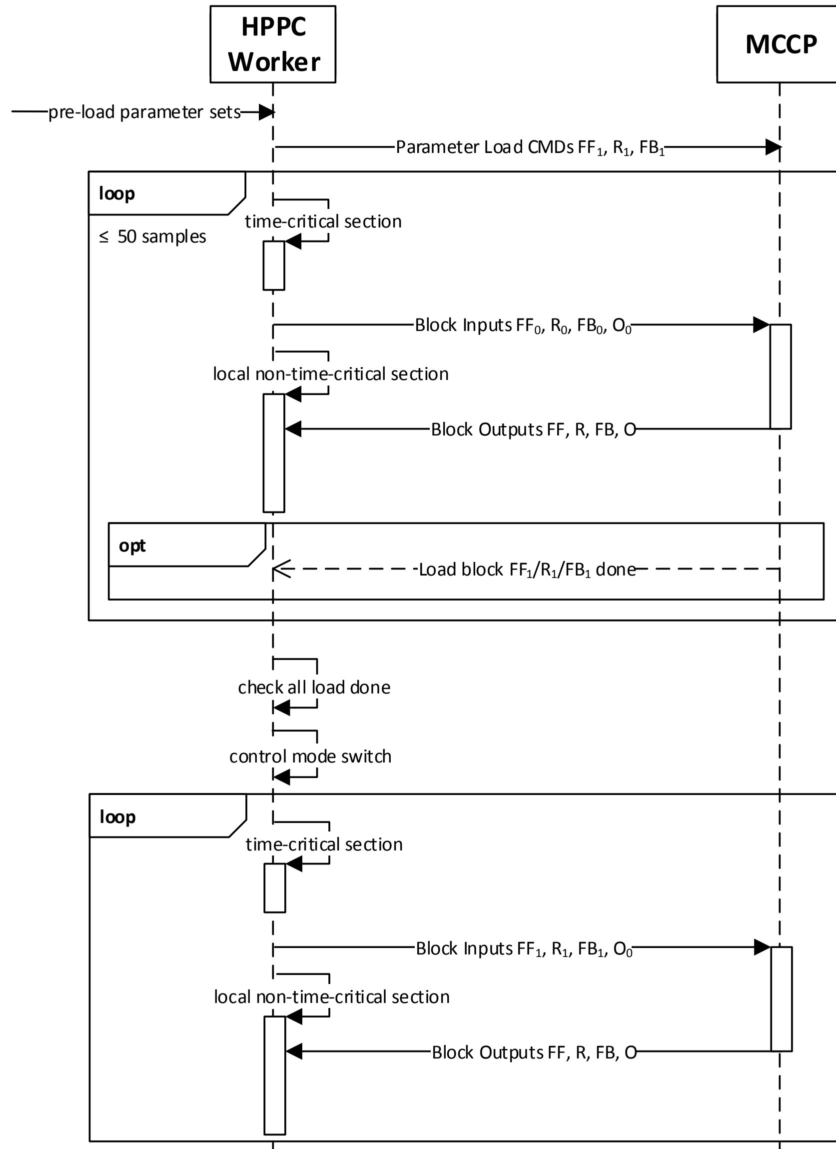


Figure 3.6: MSC of a run-time scenario of a worker with MCCP showing a control mode switch

4 | MCCP Proof of Concept Interface

This chapter describes the proof of concept interface of the MCCP. The MCCP interface has to support all necessary functionality found in Chapter 3 via memory mapped RapidIO. For implementation simplicity, the proof of concept only supports regular NWRITE operations for inbound data. Outbound data is sent by NWRITE operations as well, as shown in Figure 4.1. The address of the NWRITE encodes the type of its data as depicted by Communication Address Space.

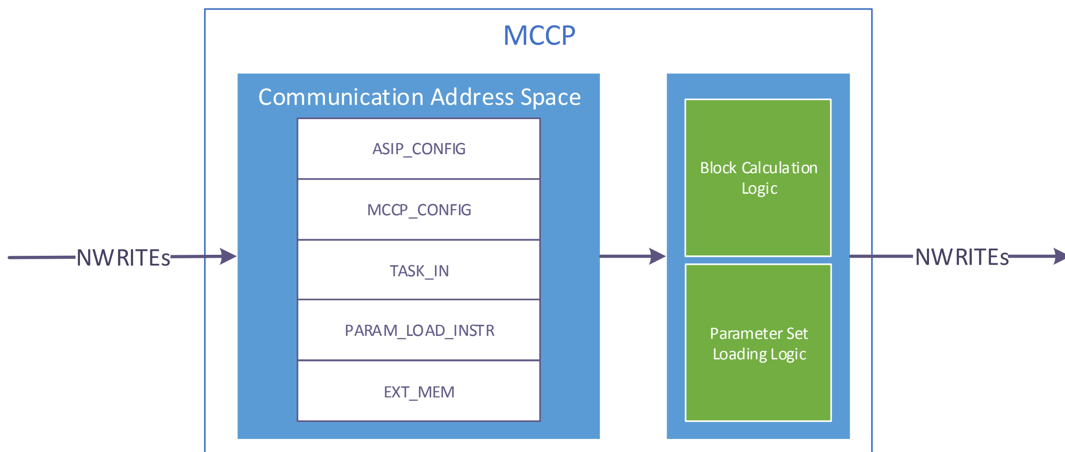


Figure 4.1: MCCP Conceptual Interface

The following functionality is supported:

- Configuring the ASIP Platform (Section 4.1)
- Configuring the MCCP (Section 4.2)
- Perform task calculations for specific inputs to generate task outputs (Section 4.3 and Section 4.4)
- Loading of parameter sets from the external memory to ASIP memory (Section 4.5)

4.1 Programming the ASIP Platform

A code generator described in [10] reads files describing the ASIP platform and the tasks mapped onto it to generate a set of configuration memory values with corresponding addresses for the ASIP platform. The generated file contains program instructions, variables, and constants for each task. The addresses are defined in the configuration address space of the ASIP platform.

For the proof of concept, these tasks are two SS_220 `pre_calc` blocks with a shared state variable ($\vec{x}_k$) for each SS_220 block in the benchmark application. E.g. the SS_220 `pre_calc` block FF is instantiated as:

```
block ss_220_FF_pre0     ss_pre_shared_state(11,11,220,sFF)
block ss_220_FF_pre1     ss_pre_shared_state(11,11,220,sFF)
```

Where `sFF` is the name of the shared state variable. Each block's tasks are mapped to a separate VPE for increased parallelism.

To program the ASIPs, the values generated by the code generator have to be written to the corresponding addresses on the configuration interface of the ASIP platform. The configuration interface of the ASIP platform is mapped to ASIP_CONFIG of the inbound memory space of the MCCP, hence the ASIP platform can be configured by NWRITEs of the values to their specified addresses with an offset of the ASIP_CONFIG base address.

4.2 MCCP Configuration

In addition to the ASIP platform configuration, the MCCP has some registers that need to be configured before task outputs can be sent back, and parameter sets can be loaded from the external memory to a task.

- TARGET_RIO_ID: RapidIO id of the intended receiver of MCCP output.
- T_O_ADDR: Base address of a writeable 32-bit memory space for task output values on the target with RapidIO id TARGET_RIO_ID.
- TO_FLAG_ADDR: Base address of a writeable 32-bit memory space for task output flags on the target with RapidIO id TARGET_RIO_ID.
- PLOAD_FLAG_ADDR: Base address of a writeable 32-bit memory space for parameter load done flags on the target with RapidIO id TARGET_RIO_ID.

Furthermore, a lookup table PARAM_LOAD_LUT with an ASIP platform configuration addresses for each task has to be filled. Each task has an entry which should contain the base address of its parameter set in the ASIP platform configuration memory space, hence these values should be generated by the code generator. For this proof of concept, they are gathered manually.

4.3 Task Input

Task inputs are written to the TASK_IN memory space by NWRITE operations. Task input parameters have a static external address uniquely identifying the input which is known by the worker. E.g. SS_220 `pre_calc` block FF has external input addresses [0..10] for the first task, and [64..74] for the second task.

For writing task inputs to the MCCP, the following conditions apply:

- Task inputs are written to their external 32-bit word address as offset of TASK_IN
- Input values are in big-endian IEEE 754 floating point format ([11])

Since the addresses used in RapidIO are byte addresses, and input values are 32-bit, the external addresses of the inputs have to be multiplied by 4. E.g. to send the input vector $[0.1, 0.2, ..., 1.1]$ to the first task of FF, thereby triggering its calculation, the data should be written as shown in Table 4.1.

Address	Value	Decimal Representation
TASK_IN+0 ∗ 4	0x3dcccccd	0.1
TASK_IN+1 ∗ 4	0x3e4ccccd	0.2
TASK_IN+2 ∗ 4	0x3e99999a	0.3
TASK_IN+3 ∗ 4	0x3ecccccd	0.4
TASK_IN+4 ∗ 4	0x3f000000	0.5
TASK_IN+5 ∗ 4	0x3f19999a	0.6
TASK_IN+6 ∗ 4	0x3f333333	0.7
TASK_IN+7 ∗ 4	0x3f4ccccd	0.8
TASK_IN+8 ∗ 4	0x3f666666	0.9
TASK_IN+9 ∗ 4	0x3f800000	1.0
TASK_IN+10 ∗ 4	0x3f8ccccd	1.1

Table 4.1: Task Input Example for FF (32-bit big-endian words)

Note that this can be done in a single NWRITE with a payload of five double-word values and one single-word value to TASK_IN_BUF+0∗4, because the external addresses are chosen such that they are successive numbers.

4.4 Task Output

Task outputs are written from the MCCP to the RapidIO id given by the TARGET_RIO_ID register. Similar to task inputs, task outputs have a static external address uniquely identifying the output.

Output values are written by NWRITEs operations to the address in T_O_ADDR with a 32-bit word offset of the value's external address. After an output value has been written, a value of 1 is written to the address in T_O_FLAG_ADDR with a 32-bit word offset of the value's external address.

For receiving task outputs from the MCCP, the following conditions apply:

- TARGET_RIO_ID contains the RIO id of the target (HPPC)
- T_O_ADDR contains a base address of a writeable 32-bit memory space on the target with a size of at least $max_ext_addr * 4$ bytes, where max_ext_addr is the maximum external address of an output
- T_O_FLAG_ADDR contains a base address of a writeable 32-bit memory space on the target with a size of at least $max_ext_addr * 4$ bytes, where max_ext_addr is the maximum external address of an output

- Each offset from T_O_FLAG_ADDR that corresponds to an external output address for which an output is expected, is set to 0

For example, for

- output value o with external address id
- value of TARGET_RIO_ID rio_id
- value of T_O_ADDR $addr$
- value of T_O_FLAG_ADDR $flag_addr$

o is written by an NWRITE to RapidIO id rio_id at address $addr + (id * 4)$. After the value has been written, a value of 0x00000001 is written by an NWRITE to RIO id rio_id at address $flag_addr + (id * 4)$.

Note that packing multiple output values together is expected to decrease overall task output delays and also the amount of computation necessary for synchronisation of outputs at the GPP. However, this would require some programmable component in order to be able to handle the general case of which values should be packed together. This is left for future work.

4.5 Parameter Set Loading

To load parameter sets from the external memory, mapped to EXT_MEM, to a task's parameter memory, Parameter Set Load instructions can be issued. Parameter Set Load instructions are written to PARAM_LOAD_INSTR by NWRITE operations. Similar to task input and output values, the task itself has a static external address uniquely identifying it.

For writing Parameter Set Load instructions to the MCCP, the following conditions apply:

- TARGET_RIO_ID contains the RIO id of the target (HPPC)
- Data written to PARAM_LOAD_INSTR holds one or multiple consecutive instructions frames of PARAM_LOAD_frame (Table 4.2)
- A valid ASIP platform configuration address is stored in PARAM_LOAD_LUT for each external task address (EXT_ADDR) in a PARAM_LOAD_frame.
- Writing of data has to start at the PARAM_LOAD_INSTR base address
- PLOAD_FLAG_ADDR contains a base address of a writeable 32-bit memory space on the target with a size of at least $max_ext_task_addr*4$ bytes, where $max_ext_task_addr$ is the maximum external task address
- Each offset from PLOAD_FLAG_ADDR that corresponds to an external task address for which a PARAM_LOAD instruction is sent, is set to 0

For each PARAM_LOAD_frame, a flag is written back to the target with RapidIO id TARGET_RIO_ID by an NWRITE operation to verify that the instruction has finished executing on the MCCP. The flag has a value of 0x00000001 and is written to the address in PLOAD_FLAG_ADDR with the task's external address as 32-bit word offset.

Relative word addr.	Bit field	Mnemonic	Description
0		PARAM_LOAD_HEADER	Parameter load instruction header
	[31:4]	nr_values	number of values to load
	[3:0]	PARAM_LOAD_OP	PSU load instruction op-code. Value: 0x2)
1	[31:0]	FROM_ADDRESS	Start external memory address (relative to the EXT_MEM base address) of values to load
2	[31:0]	EXT_ADDR	Address of the destination task

Table 4.2: PARAM_LOAD_frame

5 | MCCP Proof of Concept Design

In this chapter, the FPGA design that implements the interface as specified in Chapter 4 is given. The top-level design is shown in Figure 5.1.

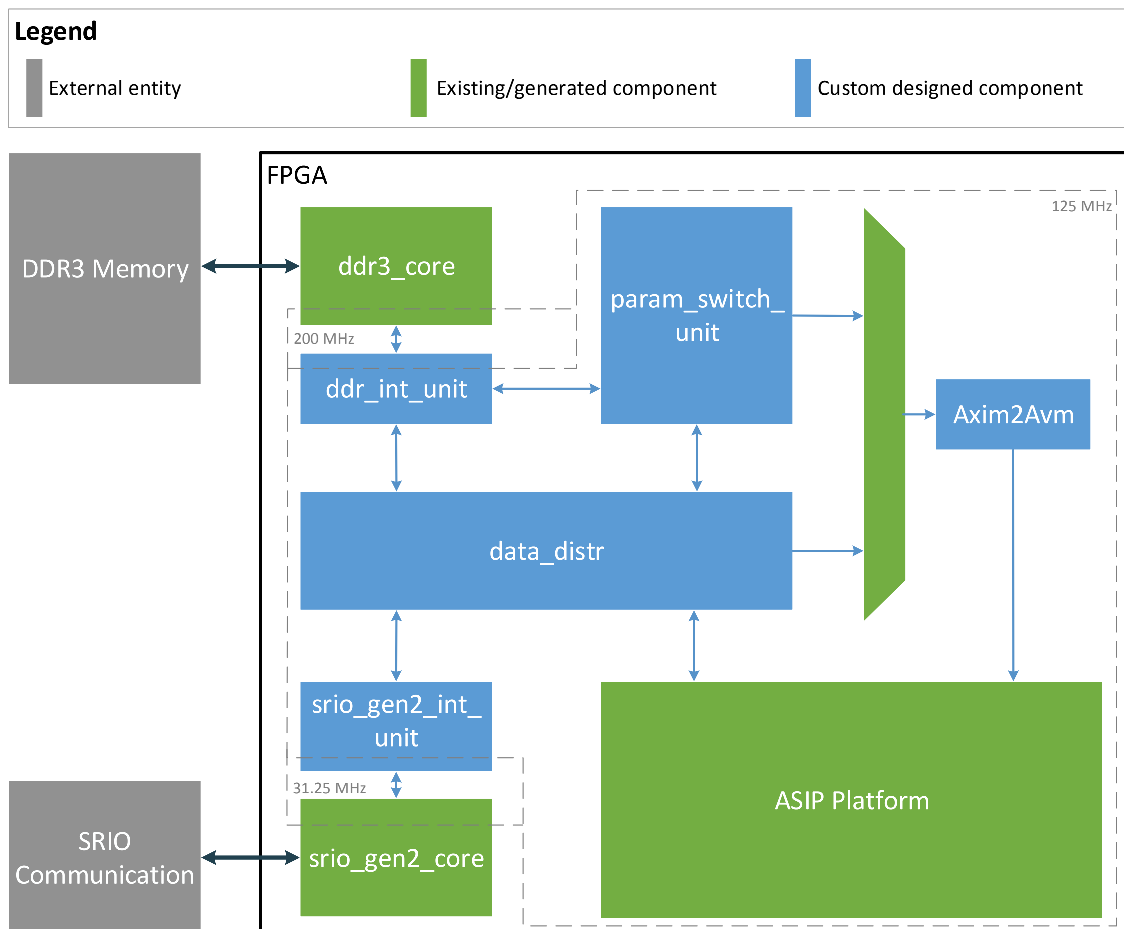


Figure 5.1: MCCP Design

The design is made for, and implemented on, a Xilinx Virtex-7 FPGA VC707 Evaluation Kit ([12]). The VC707 kit has DDR3 memory which is used as the external memory of the MCCP. It is connected to the SRIO network via its SFP+ transceivers with a single-lane 2.5 GBaud link. The main clock domain has a frequency of 125 MHz to avoid having to solve time-consuming timing problems during implementation. The design contains

- srio_gen2_core: a LogiCORE IP Serial RapidIO Gen2 Endpoint v3.1 core ([13]) for translating the transceiver signals from and to RapidIO packets

- ddr3_core: a LogiCORE IP DDR3 core ([14]) for storing data to and loading data from the DDR3 memory. Generated with a 32-bit interface width to match the ASIP Platform's configuration interface data width.
- srio_gen2_int_unit and ddr_int_unit: convert from respectively the srio_gen2_core interface and ddr3_core interface to an interface that is used internally in the main clock domain in order to keep the design modular
- data_distr: distributes the inbound data to the other components, and gathers outbound data from the other components
- param_switch_unit: handles Parameter Set Load instructions
- Axim2Avm: converts the internally used interface to an interface compatible with the ASIP Platform configuration interface. An arbiter arbitrates between param_switch_unit and data_distr.
- ASIP Platform: the ASIP platform as specified in Section 3.1

Inbound NWRITEs are always passed from srio_gen2_core to srio_gen2_int_unit. srio-_gen2_int_unit converts the NWRITEs to internally used write interfaces and passes them to data_distr. data_distr then decodes the destination of the message from the address:

- NWRITEs to ASIP_CONFIG are written to the ASIP platform's configuration interface via Axim2Avm as shown in Figure 5.2.
- NWRITEs to MCCP_CONFIG are handled as shown in Figure 5.3. If the address is equal to one of the MCCP configuration registers, the data is stored in data_distr . If the address is in PARAM_LOAD_LUT, it is written to psu_lut in param_switch_unit.
- NWRITEs to TASK_IN are passed to the ASIP Platform's external interface as shown in Figure 5.4. The ASIP platform starts a tasks computation once all inputs for that task have arrived. data_distr creates two outbound NWRITEs for each output value from ASIP Platform's external interface.
- NWRITEs to PARAM_LOAD_INSTR are passed to param_switch_unit, where they are decoded and carried out as shown in Figure 5.5.
  param_switch_unit retrieves the DDR address from the instruction, and retrieves the ASIP platform configuration address from psu_lut using the external task address in the instruction. Multiple reads from the DDR memory could be carried out before the Parameter Set Load instruction has been completed (Figure 5.6).
- NWRITEs to EXT_MEM are written to the DDR memory through ddr_int_unit as shown in Figure 5.7.

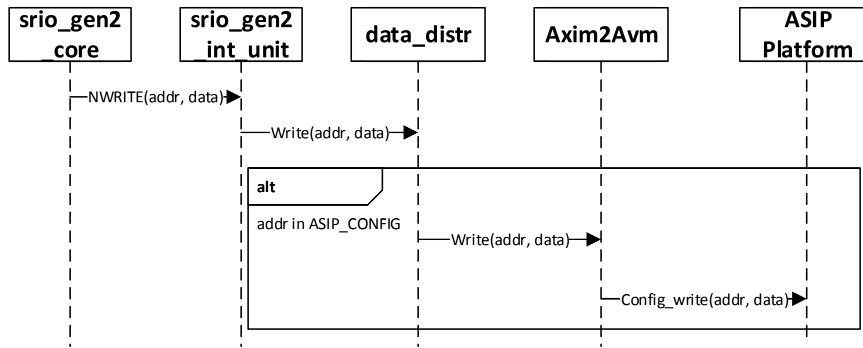


Figure 5.2: MSC for NWRITEs to ASIP_CONFIG

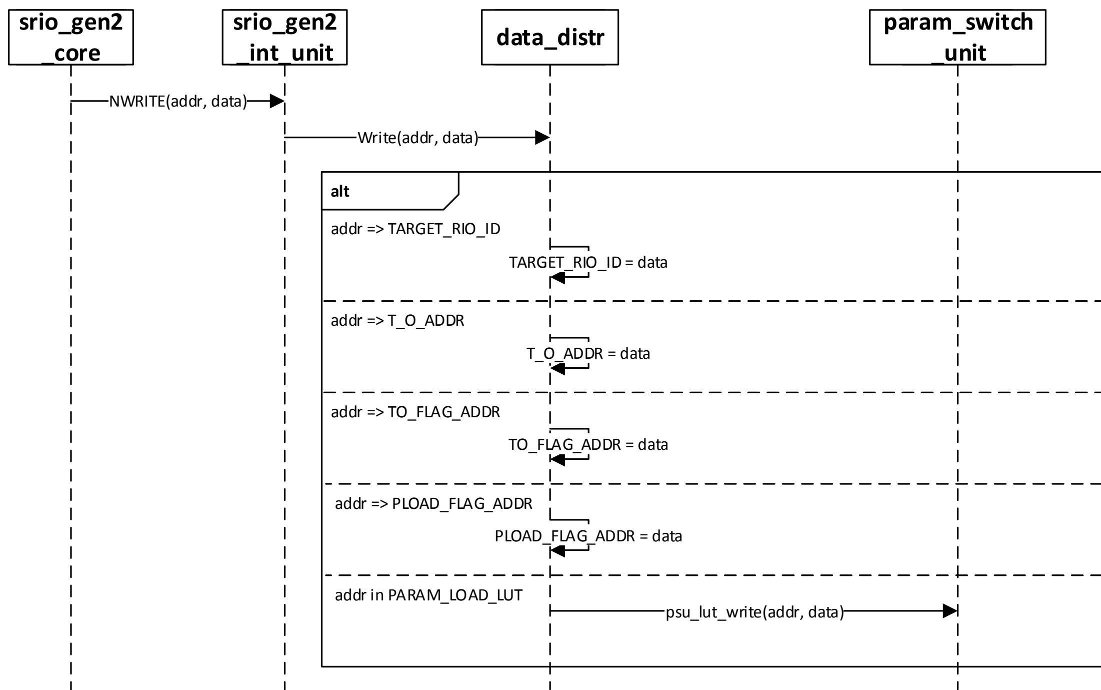


Figure 5.3: MSC for NWRITEs to MCCP_CONFIG

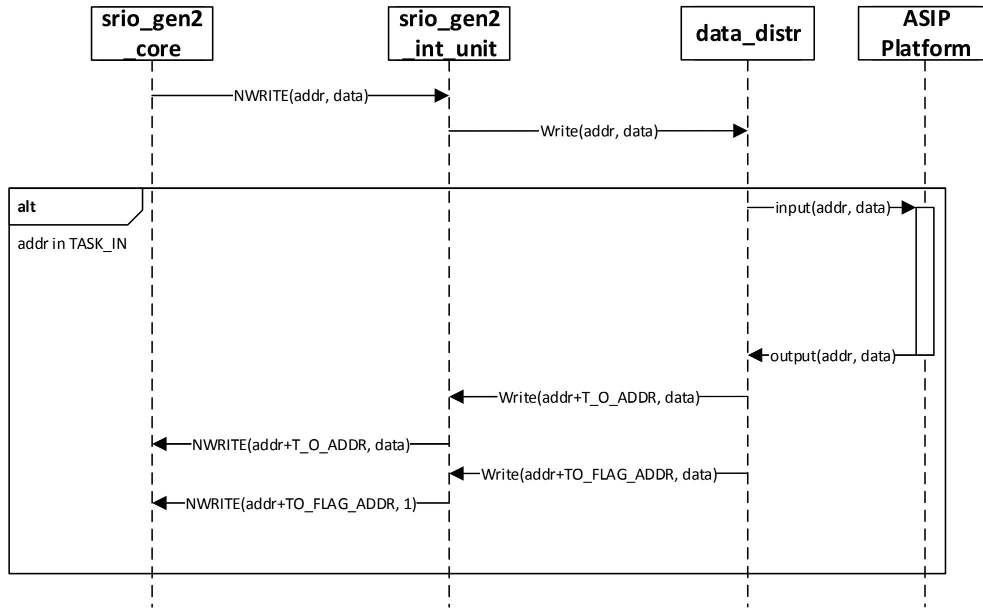


Figure 5.4: MSC for NWRITEs to TASK_IN

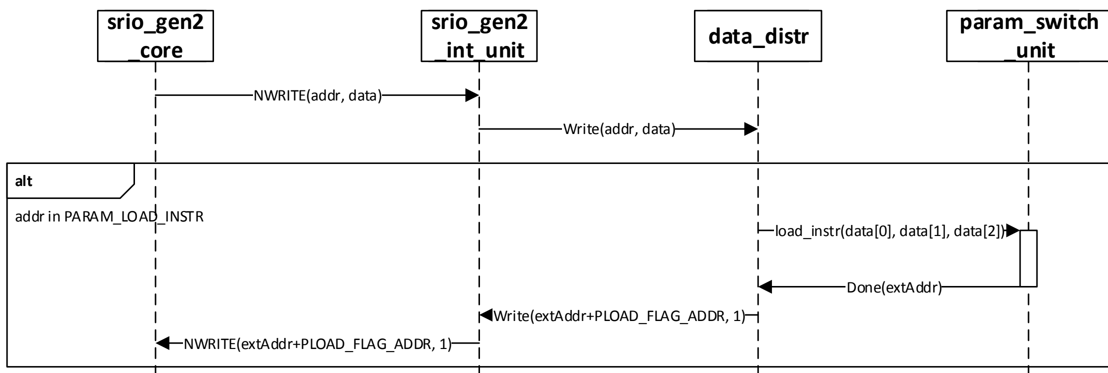


Figure 5.5: MSC for NWRITEs to PARAM_LOAD_INSTR (1 of 2)

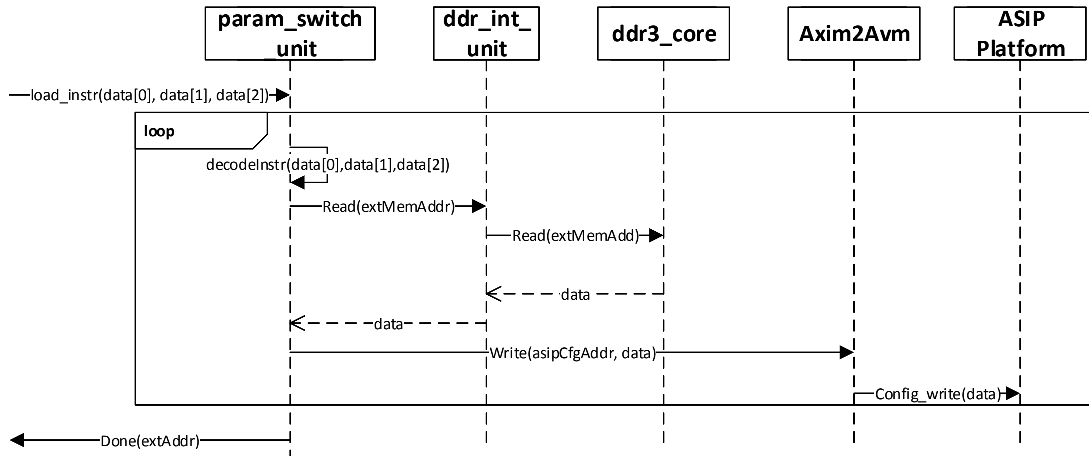


Figure 5.6: MSC for NWRITEs to PARAM_LOAD_INSTR (2 of 2)

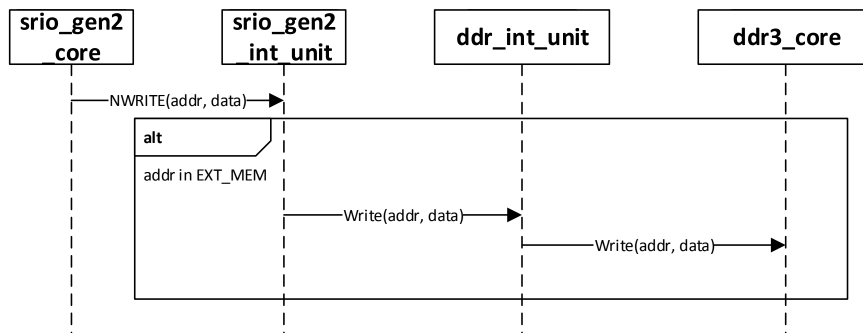


Figure 5.7: MSC for NWRITEs to EXT_MEM

6 | Experimental Evaluation

In this Chapter, the experimental evaluation of the MCCP is discussed.

- Section 6.1 describes the test setup used for evaluation
- Section 6.3 describes the metrics to be gathered
- Section 6.4 and Section 6.5 describe the benchmarking approach for the three types of metrics
- Section 6.6 lists and analyses the results of the experiments
- Section 6.7 gives conclusions of the analysis and assesses performance of the MCCP with some low-effort optimisations

6.1 Test Setup

The MCCP will be tested in a setup shown in Figure 6.1. The test setup consists of a PRIOC-QA ATCA carrier blade with

- Two PPA8548 AMC cards
- PGEA for SRIO breakout
- Xilinx VC707 FPGA Evaluation Kit
- A Linux Server to program the PPA8548 cards and FPGA
- QSA (QSFP+ to SFP+ conversion)
- RapidIO Logic Analyzer

The HPPC-worker is deployed on a PPA8548 AMC card plugged in to a PRIOC-QA ATCA carrier blade. The MCCP will be deployed on a Xilinx Virtex 7 (VC707) FPGA development kit which is not an AMC card, hence a PGEA SRIO breakout AMC card with a QSA will be used. The QSA converts the QSFP+ connector of the PGEA to an SFP+ connector which is connected to the FPGA. Note that the SFP+ connector only connects a single SRIO lane, hence the connection form the FPGA to the switch is single lane, while the connections from the PPAs to the switch are quad-lane (four times as fast).

The Logic Analyzer is used to monitor RIO traffic from and to the MCCP. It can be connected between the switch and the MCCP or between a PPA and the switch.

6.2 Software

The Linux Server is used to program the FPGA. It runs Debian 3.2.54-2 x86_64. The Linux server runs a service called Xilinx Hardware Server to which a machine running Xilinx Vivado

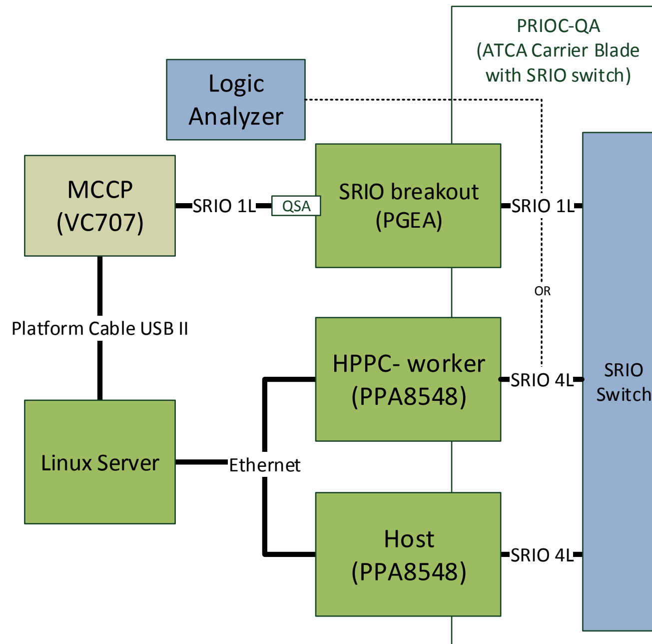


Figure 6.1: Test setup hardware

can connect. To measure the internal FPGA timings, Integrated Logic Analyzer (ILA) probes are used ([15]).

The PPA8548 boards run Wind River Linux glibc_cgl (preempt_rt) 2.0.

6.3 Benchmarks and Metrics

The focus of the benchmarks will be on offloading the state space block 220 pre-calculation (SS_220_pre), since this is expected to give the most performance gain for the benchmark application (Chapter 3). The benchmark application contains four SS_220 blocks, where two types can be distinguished:

- 11 inputs, 11 outputs (SS_220_11_11: FF, R, FB)
- 12 inputs, 11 outputs (SS_220_12_11: O)

Three different kinds of metrics will be measured:

- Delays of the MCCP (Section 6.3.1)
- Resource utilisation of the MCCP deployed on an FPGA (Section 6.4.3)
- Correctness of outputs (Section 6.5)

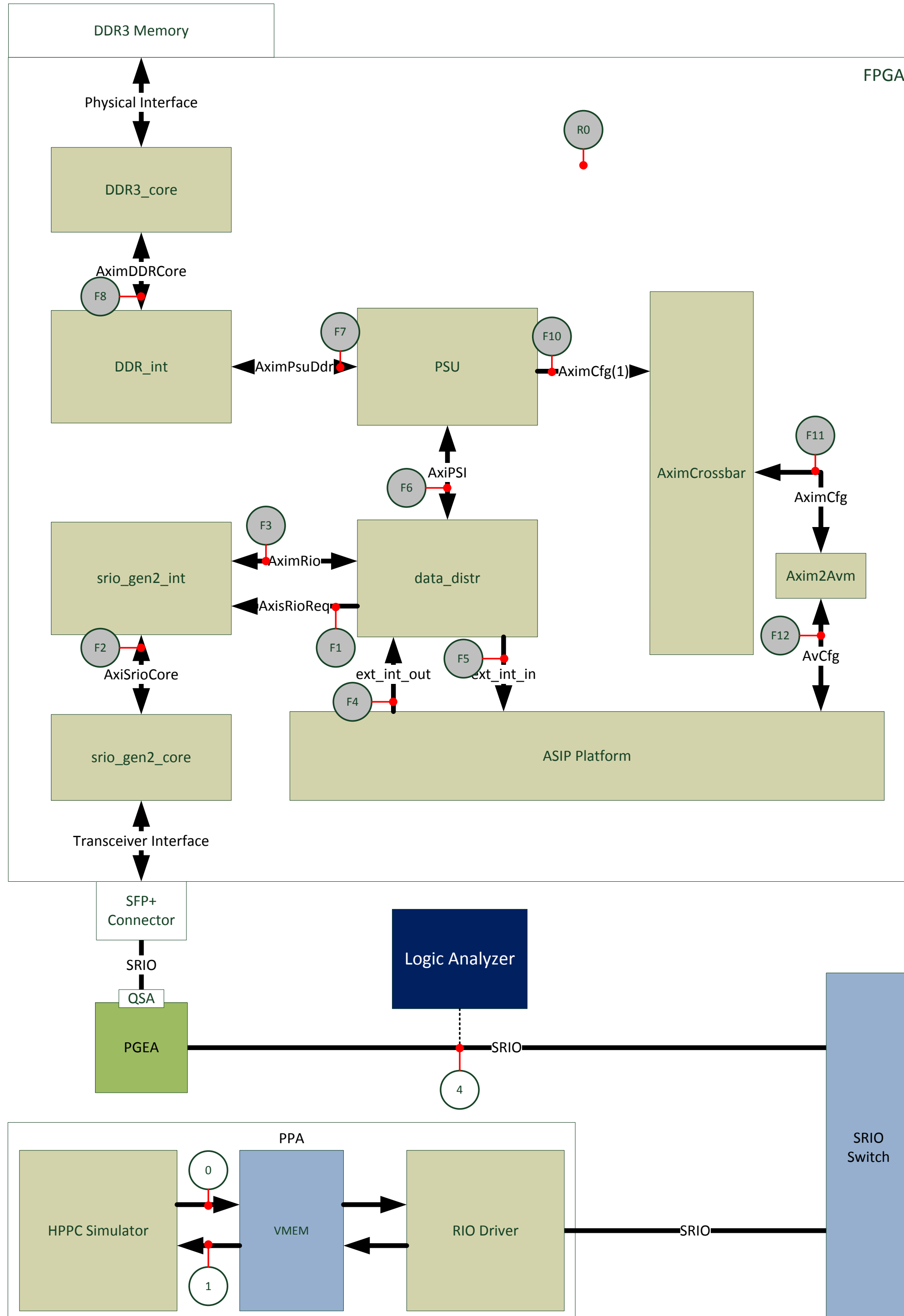


Figure 6.2: Measuring points

Figure 6.2 shows the measuring points in the test setup. Measuring points prepended with an 'F' are probed using ILA cores in the FPGA design. Other measuring points are

- ⟨0⟩: Memory write probe. Used to measure the start of sending task inputs or PSU instructions.
- ⟨1⟩: Memory read probe. Used to measure the end of receiving task outputs or PSU done flags and correctness of results.
- ⟨4⟩: Logic Analyzer probe. Used to measure MCCP input-to-output times.
- ⟨R0⟩: FPGA resource utilisation.

6.3.1 Delays

Input-to-output delays of two different MCCP run-time functionalities should be measured in order to be able to estimate the maximum sample frequency:

- Task input to output
- Parameter Set loading

Both of these can be measured at three points:

- Internally: input at ⟨F2⟩ to output at ⟨F2⟩ (Section 6.4)
- Externally: input at ⟨4⟩ to output at ⟨4⟩ (Section 6.4.4)
- Completely: input at ⟨0⟩ to output at ⟨1⟩ (Section 6.5)

6.4 Internal Benchmarks Approach

Internal delays are measured using ILA cores. ILA cores can be inserted after synthesis in Vivado. They are inserted on each of the signal groups marked by a probe in Figure 6.2.

The total internal input at ⟨F2⟩ to output at ⟨F2⟩ can be decomposed in multiple smaller delays. The decomposition differs depending on the type of input as explained in Sections 6.4.1 and 6.4.2.

6.4.1 Task Input-to-Output Delay

Internal task Input-to-Output delay is measured at ⟨F2⟩. It is composed of

- ⟨F2⟩ → ⟨F3⟩
- ⟨F3⟩ → ⟨F5⟩
- ⟨F5⟩ → ⟨F4⟩
- ⟨F4⟩ → ⟨F1⟩
- ⟨F1⟩ → ⟨F2⟩

The time it takes to execute the state space calculations SS_220_11_11_pre and SS-_220_12_11_pre (⟨F5⟩ → ⟨F4⟩) can be calculated based on the code generated for the ASIP Platform and its clock frequency (125 MHz). The ASIP Platform adds some additional delay for routing the data from and to the external interface of the ASIP Platform.

The total internal task input-to-output delay is defined as the time it takes from the moment the first packet of the inputs for one task arrives at ⟨F2⟩ to the time of the last beat of the last output packet of that task is accepted by the srio_gen2_core at ⟨F2⟩. All sub-delays are measured with a similar definition, i.e. beat of first at the input to beat of last at output side.

6.4.2 PSU Parameter Load Delay

Once a parameter load instruction arrives at ⟨F2⟩, they go to the PSU which handles the instruction:

- ⟨F2⟩ → ⟨F3⟩
- ⟨F3⟩ → ⟨F6⟩

Then, the load instruction is carried out. It is broken up into one or more DDR loads of 256 words that follow a path:

- ⟨F7⟩ → ⟨F8⟩
- ⟨F8⟩ → DDR3 Memory
- DDR3 Memory → ⟨F8⟩
- ⟨F8⟩ → ⟨F7⟩
- ⟨F7⟩ → ⟨F10⟩
- ⟨F10⟩ → ⟨F11⟩
- ⟨F11⟩ → ⟨F12⟩

When all loads have been carried out, an NWRITE request for the 'instruction done' flag goes back through:

- ⟨F6⟩ → ⟨F1⟩
- ⟨F1⟩ → ⟨F2⟩

The total internal PSU parameter load delay is defined as the time it takes from the moment the first packet of a parameter load instruction for one task arrives at ⟨F2⟩ to the time of the last beat of the load done packet is accepted by srio_gen2_core at ⟨F2⟩.

The DDR load delay (⟨F7⟩ → ⟨F12⟩) is defined as the time it takes from the moment the read request of 256 32-bit words is presented at ⟨F7⟩ to the time of the last beat of the read data is accepted at ⟨F12⟩.

6.4.3 FPGA Resource Utilisation

Resource utilisation ⟨R0⟩ is composed of the resource usage of all blocks. For each block, the following metrics are gathered:

- Number of Slice LUTs
- Number of Slice Registers
- Number of Block Rams (BRAM)
- Number of DSPs

6.4.4 External Benchmarks Approach

With the Logic Analyzer, the traffic on the RIO link between the switch and the PGEA can be monitored. From the Logic Analyzer measuring point (⟨4⟩), inbound packets are subject to several delays before arriving at the first internal measuring point ⟨F2⟩:

- RX PGEA delay
- RX QSA delay
- RX srio_gen2_core delay

Outbound RIO packets from ⟨F2⟩ are subject to other delays:

- TX srio_gen2_core delay
- TX QSA delay
- TX PGEA delay

Task input packets can be matched to corresponding task output packets, and parameter load instruction packets can be matched to the corresponding packets with 'load done' flags using their addresses. The total time from task input to task output or parameter load instructions can then be calculated using the timestamps provided by the Logic Analyzer.

The total external task input-to-output delay is defined by the time it takes from the moment the first packet of the inputs for one task arrives at ⟨4⟩ to the time of the last beat of the last output packet of that task arrives at ⟨4⟩.

The total external PSU parameter load delay is defined by the time it takes from the moment the first packet of a parameter load instruction for one task arrives at ⟨4⟩ to the time of the last beat of the load done packet arrives at ⟨4⟩.

6.5 HPPC Simulator Benchmarks Approach

An HPPC simulator application (Test Application in Figure 6.2) running on the PPA8548 maps outbound and inbound virtual memory to and from the MCCP (depicted by VMEM in Figure 6.2). Data written to the outbound memory is sent to the MCCP by NWRITEs, and

NWRITEs arriving from the MCCP are written to the inbound memory. From the point of view of the MCCP, it acts as a real HPPC running a servo application with offloaded blocks on the MCCP.

Using VMEM, the test application first initialises the MCCP by sending the ASIP Platform configuration data and setting the necessary MCCP specific registers (as specified in Chapter 4).

To benchmark the MCCP, the test application sends inputs for a predefined scenario to the MCCP at rates ranging from 20 kHz to 50 kHz with 10 kHz increments, or up until the maximum achievable sample frequency if lower than 50 kHz. The scenarios contain regular task inputs for each sample for all offloaded blocks, and control mode switch (i.e. parameter load instruction) sample numbers. The scenarios are defined in a file which is parsed and loaded into memory by the HPPC simulator before execution starts. Scenarios run for 10^6 samples.

The following events are logged with a timestamp in a file that contains the VHDL and test application version information:

- Writing of block inputs to VMEM
- Reading of block outputs from VMEM
- Writing of PSU load instructions to VMEM
- Reading of PSU load done flags from the VMEM

The values are read from the memory as early as possible, that is, as soon as it is present in the local memory.

Two different offloading scenarios are benchmarked:

- One SS_220_11_11_pre block
- All four SS_220_pre blocks of the benchmark application

For the maximum sample frequency reached for any application, it will be manually determined what component (e.g. the RapidIO link or the MCCP internally) causes the bottleneck.

6.5.1 Correctness

The benchmark scenarios are executed completely on the PPA first to obtain a baseline version of the expected outputs. Correctness could be validated by comparing all block outputs of a run to a baseline version. However, because of some unknown issue, the VPEs return faulty output values for the SS_220_pre blocks, even in simulation. Therefore, the correctness is not validated.

6.5.2 Delay

Input-to-output delays can be calculated from the log files. For both the block input-to-output and the parameter load delays, an average, maximum, and minimum are gathered.

Note that delays cannot be measured simultaneously. Consider for example task outputs and corresponding flags arriving shortly before a parameter load done flag. All task output flags first have to be read to ensure that the all output values have arrived. After this, the parameter load done flag can be checked, but now a part of the measurement time for the task output flags is added to the measured delay for the parameter set load.

To avoid this, the scenarios are first executed in the most realistic manner:

- 50 samples before a control mode switch, the parameter set load instructions are sent
- Block IO is continued during the execution of the parameter set load and all block output flags are checked. Delays of the first block are logged.
- At the control mode switch sample, the parameter set load done flags are synchronised

This ensures that the MCCP works as expected and is functionally correct. After this, the scenario is executed again to measure parameter set load delays:

- At the time of a control mode switch, the parameter set load instructions are sent and immediately synchronised by busy waiting so their actual delay can be logged.
- Block IO is only done in between and the output flags are checked. Delays of the first block are logged.

The delays of the first execution can be compared to the delays of the second one to see if the parameter set loads have any significant impact on the block IO delays. Finally, for the four block offloading case, the task input-to-output delays of the other blocks can be measured in three more separate executions of the scenario by synchronising the block under test first.

6.6 Results and Analysis

In this section, the experimental evaluation results are listed and analysed. Section 6.6.1 lists the resource utilisation of the MCCP design on the FPGA. Section 6.6.2 lists the internally and externally measured detailed timing metrics both from simulation and deployment. Section 6.6.3 lists the metrics measured from the HPPC simulator application.

6.6.1 Resource Utilisation

Table 6.1 shows the resource utilisation metrics for the ASIP platform and total design. It has one column for the metrics after synthesis, and one for the metrics after implementation of the design. The complete resource utilisation table for all components can be found in Appendix B.

The FPGA on the Xilinx VC707 development kit has a total of

- 303600 LUTs
- 607200 Flip-Flops/Registers
- 1030 36Kb BRAMs (each block can be utilised as two 18 Kb BRAMs)
- 2800 DSP48s

Block	Synthesis	Implementation
ASIP Platform		
#Slice LUTs (%)	61398 (20.22%)	50761 (16.72%)
#Slice Registers (%)	76983 (12.68%)	67325 (11.09%)
#BRAMs (%)	709 (68.83%)	709 (68.83%)
#DSPs (%)	256 (0.00%)	256 (0.00%)
Total		
#Slice LUTs (%)	87521 (28.8%)	75010 (24.71%)
#Slice Registers (%)	102573 (16.89%)	90924 (14.97%)
#BRAMs (%)	718 (69.71%)	718 (69.71%)
#DSPs (%)	256 (9.14%)	256 (9.14%)

Table 6.1: MCCP Resource Utilisation Table

The resource utilisation metrics are as expected: ASIP Platform uses most of the BRAMs and all DSPs in the design. On this particular FPGA, Apart from the BRAMs, utilisation of the other resources in the complete design is relatively low, which allows for more logic around or inside the ASIP Platform. There are enough unused resources left to add more ASIPs to the ASIP Platform, or increase the sizes of the current ASIPs, although only one VPE of the same size could be added.

6.6.2 MCCP Timing Metrics

Table 6.2 shows the timing metrics of the input-to-output delays measured internally in the FPGA and with the Logic Analyzer for just one offloaded block. Due to limitations of the test setup, these metrics are obtained with inbound NWRITEs containing at most one 32-bit word. Therefore, task input arrives in 11 or 12 separate NWRITEs, and a parameter load instruction arrives in 3 NWRITEs.

In the simulation column, the external Task Input-to-Output delay and PSU Parameter Loading Delay are out of scope, and hence left empty. The srio_gen2_core and ddr3_core are replaced by VHDL simulator programs, so the DDR read latency is not taken into account during simulation.

In the Measured (Deployed) column, several rows are left empty because they crossed into another clock domain. Although not impossible to measure these delays using ILA cores, it is very time-consuming to set up and does not necessarily add to the big picture of the results.

Metric	Simulation	Measured (Deployed)
SS_220_11in_11out_pre Execution time	22.02 μs	22.02 μs
SS_220_12in_11out_pre Execution time	22.02 μs	22.02 μs
Task I/O: $\langle$F2$\rangle \rightarrow \langle$F3$\rangle$	1.03 μs	-
Task I/O: $\langle$F3$\rangle \rightarrow \langle$F5$\rangle$	0.99 μs	1.48 μs
Task I/O: $\langle$F5$\rangle \rightarrow \langle$F4$\rangle$	23.58 μs	24.06 μs
Task I/O: $\langle$F4$\rangle \rightarrow \langle$F1$\rangle$	2.56 μs	2.02 μs
Task I/O: $\langle$F1$\rangle \rightarrow \langle$F2$\rangle$	2.74 μs	-
Internal Task Input-to-Output Delay	25.76 μs	25.76
External Task Input-to-Output Delay	-	28.75 μs
Parameter Load: $\langle$F2$\rangle \rightarrow \langle$F3$\rangle$	0.50 μs	-
Parameter Load: $\langle$F3$\rangle \rightarrow \langle$F6$\rangle$	0.54 μs	0.42 μs

Parameter Load: $\langle F6\rangle \rightarrow \langle F7\rangle$	0.49 μs	0.39 μs
Parameter Load: $\langle F7\rangle \rightarrow \langle F8\rangle$	0.05 μs	-
Parameter Load: $\langle F8\rangle \rightarrow \langle F7\rangle$ (256 words)	6.10 μs	-
Parameter Load: $\langle F7\rangle \rightarrow \langle F10\rangle$ (256 words)	6.10 μs	6.10 μs
Parameter Load: $\langle F10\rangle \rightarrow \langle F11\rangle$ (256 words)	6.10 μs	6.10 μs
Parameter Load: $\langle F11\rangle \rightarrow \langle F12\rangle$ (256 words)	6.14 μs	6.14 μs
Parameter Load: $\langle F6\rangle \rightarrow \langle F1\rangle$	0 μs	0 μs
Parameter Load: $\langle F1\rangle \rightarrow \langle F2\rangle$	0.08 μs	-
DDR read latency: $\langle F8\rangle \rightarrow \langle F8\rangle$ (read accepted to read data valid)	-	0.16 μs
DDR read latency: $\langle F7\rangle \rightarrow \langle F7\rangle$ (read accepted to read data valid)	-	0.27 μs
Internal PSU Parameter Loading Delay	1444.64 μs	1472.75 μs
External PSU Parameter Loading Delay	-	1474.31 μs

Table 6.2: MCCP Timing Metrics

The first thing that stands out is the 0.5 μs longer Task I/O from $\langle F3\rangle \rightarrow \langle F5\rangle$ delay in the actual deployed version compared to the simulation results. Inspection with the Logic Analyzer between the switch and the MCCP yields that the arrival of one of the NWRITEs for a task input is consistently delayed by 0.5μs. Between the PPA and the switch, the 0.5μs added delay is not visible. No packet-retry control symbols are detected on both the link between the PPA and the switch, and the link between the switch and the MCCP. Therefore, this delay can be fully attributed to the internal switch behaviour when converting from the quad-lane PPA link to the single-lane MCCP link.

In general, the delay on the path from $\langle F2\rangle \rightarrow \langle F5\rangle$ could be reduced significantly if inputs would arrive in a single NWRITE: each NWRITE requires a header which encodes the type of packet and contains the address. Instead of having to process this header 11 or 12 times for one task, it could be processed only once for all tasks that are offloaded (as long as the number of 32-bit input values is less than or equal to 64). Simulation yields a delay of 0.3μs from $\langle F2\rangle \rightarrow \langle F5\rangle$ for one task if the inputs are packed into one NWRITE; less than a third of, or 0.7μs less than the delay with single-word NWRITEs on that path.

The path from $\langle F4\rangle \rightarrow \langle F2\rangle$ has a shorter delay in the deployed version compared to the simulation version. This is due to the srio_gen2_core having a higher packet acceptance rate for these numbers of NWRITE requests than the simulated version. This causes the internal task input-to-output delay to be exactly the same in simulation as in deployment, even though the path from $\langle F2\rangle \rightarrow \langle F5\rangle$ is significantly longer in deployment.

There is a difference of 3 μs between the internally and externally measured Task input-to-output delays. This means that the components for which the delay was not measured (the path from srio_gen2_core to the PGEA and QSA) adds an additional significant delay. srio_gen2_core has a latency of between 0.7 and 1.0 μs in single-lane mode, depending on the direction and type of message. This delay is expected to be decreased by a factor of more than 2 if srio_gen2_core runs in quad-lane mode.

The Parameter Load input path does not show the big difference between simulation and deployment, because the input size is only 3 words. The total Internal Parameter Loading delay is larger in deployment, which can be attributed fully to the real DDR3 read delay compared to the read delay in the simulation program.

From these measurements, we can already clearly see the bottleneck: parameter loading for

a single block already takes significantly longer than the allowed 1.25ms for a servo application running at 40 kHz with 50 samples of preloading time. Due to the ASIP Platform accepting at most one 32-bit value per 3 clock cycles on the configuration interface, loading 256 values from the DDR3 memory into the ASIP Platform takes over 6 μs.

6.6.3 HPPC Simulator Timing Metrics

Table 6.3 shows the results for single-block offloading and Table 6.4 shows the results for quad-block offloading measured at the PPA running an HPPC simulator application.

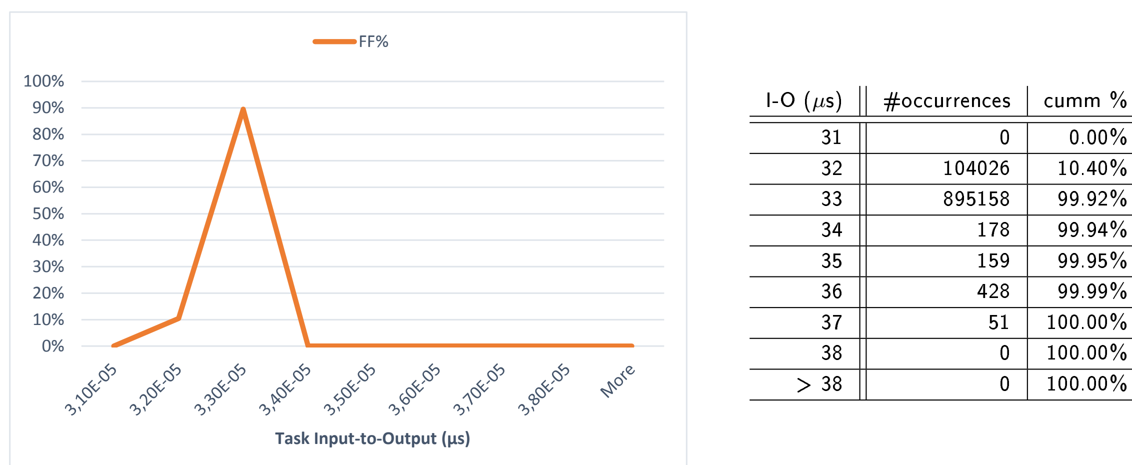
As mentioned in Section 6.5.2, the MCCP is first functionally verified by running the scenarios as realistically as possible. Input-to-output times for one block could be compared to a less realistic scenario where the parameter set loads are not executed simultaneously with block IO. It was found to have no measurable effect on the block's input-to-output delay, hence the following measurements are representative for a realistic scenario.

For the quad block offloading case, blocks inputs and parameter loads are sent in sequence FF, R, FB, O. Single block offloading only uses block FF.

Metric	Minimum Time (μs)	(Average) Time (μs)	Maximum Time (μs)
20 kHz			
SS_220_11in_11out_pre input-to-output	31.6	32.2	36.5
Parameter load instruction input-to-output	1478.3	1482.2	1483.7
30 kHz			
SS_220_11in_11out_pre input-to-output	31.5	32.2	36.8
Parameter load instruction input-to-output	1478.3	1481.5	1483.2
Maximum			
SS_220_11in_11out_pre input-to-output	31.5	32.2	37.1
Parameter load instruction input-to-output	1478.3	1481.7	1483.6

Table 6.3: PPA Delay Metrics Single SS_220

For the single offloaded block case, the metrics are consistent across different sample frequencies. The measurements show some small anomalies, but their count is minimal, as shown in Figure 6.3. The small spike between 35 and 36 μs is most likely due to some buffering behaviour in the switch, as no retry control symbol is detected on the SRIO network during execution of these benchmarks.



I-O (μs)	#occurrences	cumm %
31	0	0.00%
32	104026	10.40%
33	895158	99.92%
34	178	99.94%
35	159	99.95%
36	428	99.99%
37	51	100.00%
38	0	100.00%
> 38	0	100.00%

Figure 6.3: Task IO delay histogram for single block offloading

Metric	Minimum Time (μs)	(Average) Time (μs)	Maximum Time (μs)
20 kHz			
FF SS_220_11in_11out_pre input-to-output	32.0	33.3	39.5
FF Parameter load instruction input-to-output	1477.7	1478.6	1480.9
R SS_220_11in_11out_pre input-to-output	32.3	33.6	38.5
R Parameter load instruction input-to-output	2941.7	2948.8	2954.6
FB SS_220_11in_11out_pre input-to-output	33.5	36.0	41.6
FB Parameter load instruction input-to-output	4409.7	4419.5	4428.0
O SS_220_12in_11out_pre input-to-output	32.7	36.9	42.7
O Parameter load instruction input-to-output	5877.2	5890.4	5901.9
30 kHz			
FF SS_220_11in_11out_pre input-to-output	32.0	33.3	40.3
FF Parameter load instruction input-to-output	1477.6	1478.6	1481.5
R SS_220_11in_11out_pre input-to-output	32.4	33.6	39.5
R Parameter load instruction input-to-output	2947.0	2947.6	2948.1
FB SS_220_11in_11out_pre input-to-output	33.4	35.8	41.0
FB Parameter load instruction input-to-output	4416.7	4417.6	4420.3
O SS_220_12in_11out_pre input-to-output	32.7	36.9	44.5
O Parameter load instruction input-to-output	5886.7	5887.6	5888.5
Maximum			
FF SS_220_11in_11out_pre input-to-output	32.0	33.3	39.5
FF Parameter load instruction input-to-output	1478.1	1478.6	1481.2
R SS_220_11in_11out_pre input-to-output	32.4	33.6	39.7
R Parameter load instruction input-to-output	2947.0	2947.5	2948.1
FB SS_220_11in_11out_pre input-to-output	33.5	35.8	41.4
FB Parameter load instruction input-to-output	4416.9	4417.5	4418.5
O SS_220_12in_11out_pre input-to-output	32.4	36.9	42.5
O Parameter load instruction input-to-output	5886.9	5887.6	5888.7

Table 6.4: PPA Delay Metrics Quad SS_220

Again, the obtained input-to-output times are consistent throughout the different sample frequencies. The parameter load instruction input-to-output time is as expected: it grows linearly with the index in the input sending sequence of the blocks.

The task input-to-output times show some more interesting behaviour as shown in Figure 6.4. The input-to-output time for the second block (R), is slightly higher than the first block (FF), but the third block (FB) has an input-to-output time of on average 2 μs higher. Further inspection with the Logic Analyzer yields that the switch sends multiple packet-retry symbols to the PPA during the task input NWRITEs in this scenario. This explains the 2 μs additional delay and the increased spread of measurement values for FB and O.

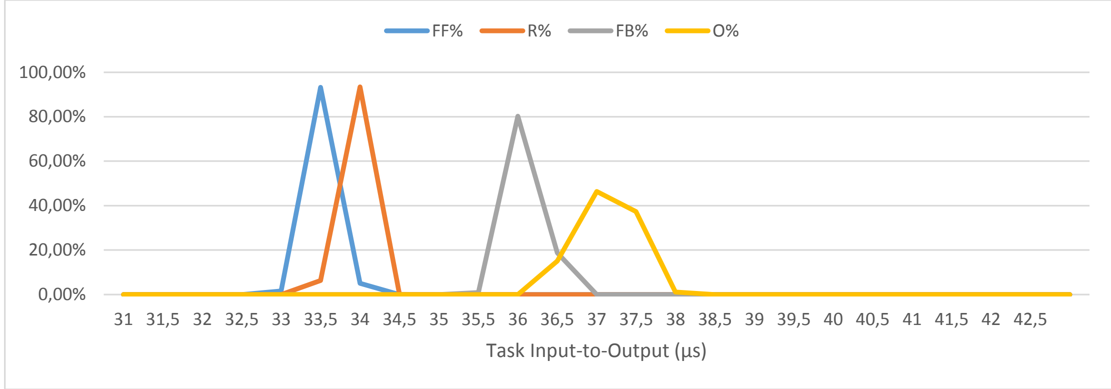


Figure 6.4: Task IO delay histogram for quadruple block offloading

I-O (μs)	#occ. FF	#occ. R	#occ. FB	#occ. O
32	11	0	0	0
33	15559	135	0	54
34	982913	998061	124	126
35	771	1014	38	101
36	204	203	811836	76
37	296	271	187419	614878
38	212	272	264	384252
39	25	37	198	119
40	8	7	106	138
41	1	0	14	207
42	0	0	1	42
$>$ 42	0	0	0	7

Table 6.5: Task IO delay data table for quadruple block offloading

6.7 Experimental Evaluation Conclusion

Results show that parameter load delays are by far the bottleneck of the MCCP. For quadruple SS_220_pre block offloading, if 50 samples are available for preloading parameter sets, the maximum sample frequency would be upper bounded by

$$\frac{50}{5887.6\mu\text{s}} = 8.5\text{kHz}$$

Assuming that parameter load delays can be reduced to load all parameter sets within 50 sample periods, results show that the maximum sample frequency of the MCCP for quadruple block offloading is bounded by the input-to-output delay of the last SS_220_pre block (O). Hence, an upper bound for the sample frequency of a worker which uses the MCCP is

$$\frac{1}{36.9\mu\text{s}} = 27.1\text{kHz}$$

However, this implementation uses an SS_220_pre calculation with a complete A matrix. According to the description of the benchmark application, A can be assumed to be upper triangular. This assumption could be used to optimise the code of the tasks. Additionally, the main clock domain, and thus the ASIP platform has a frequency of 125 MHz. If this is increased to 200 MHz, the computation time of the SS_220_pre blocks is expected to be $7\mu\text{s}$ as given in Section 2.4. This saves $22.02 - 7 = 15.02\mu\text{s}$ on the total delay. With these improvements, the maximum sample frequency is bounded by

$$\frac{1}{36.9 - 15.02\mu\text{s}} = 45.7\text{kHz}$$

.

Furthermore, using a quad-lane SRIO link instead of a single-lane link between the MCCP and the switch would significantly improve not only the network communication overhead, but is also expected to decrease the latency of srio_gen2_core through which all communication is fed. Together with the possibility to send the input data in a single NWRITE, the input-to-output delay can be easily decreased by more than $1\mu\text{s}$. Additionally, the packet-retries caused by the transition from quad-lane SRIO to single-lane are expected to be eliminated. This reduces the total delay by an additional 2 μs. The maximum input-to-output delay is brought down to

$$36.9 - 15.02 - 1 - 2\mu\text{s} = 18.9\mu\text{s}$$

while the sample frequency bound is increased to

$$\frac{1}{36.9 - 15.02 - 1 - 2\mu\text{s}} = 53.0\text{kHz}$$

From the 2 μs of added delay caused by retries, the conclusion can be drawn that interference on the communication channel can have a significant impact on the input-to-output delay. Such retries could be caused by a spike in a link's utilisation and therefore decrease the predictability of the system. Considering the hard real-time requirements in the lithography machine, network communication should be regulated such that no retries are guaranteed for the MCCP. Alternatively, a private channel could be added between each PPA-MCCP pair.

Finally, future MCCP designs could feature, as mentioned in Section 4.4, an output packer which avoids having to check many flags for each task. It would be possible to pack all four of the task outputs in a single NWRITE, and consequently only one flag has to be checked at the GPP. This also has the advantage that instead of having to create two NWRITE requests for each output value, thereby blocking new output values arriving from the ASIP platform, only one NWRITE for the values and one for the output flag has to be done.

7 | Conclusion and Future Work

Current GP architectures used for servo control in lithography machines will not be able to cope with the future development in the field. In order to tackle this challenge, a design for an FPGA motion controller co-processor (MCCP) has been proposed. As a proof of concept, an MCCP which focusses on acceleration of four large State-Space computations in a benchmarking application is implemented on an FPGA and experimentally evaluated. As calculating core, the design uses the same kind of ASIP platform as used in previous work [8].

Experimental results (Section 6.7) show that the goal sample frequency of 40kHz could be achieved, but work has to be done to support the complete case of ASML motion controllers. I.e. the ASIP platform was not able to preload parameter sets from the DDR3 memory into the blocks fast enough over its configuration interface. If parameter sets cannot be preloaded fast enough, then control mode switches need a significantly longer preloading time, which decreases the degree of direct control over the servo application. Therefore, the main recommendation is to create a special parameter loading interface, or restructure the configuration interface of the ASIP platform such that it will be able to load data much faster and in a more scalable manner than it is currently able to.

Other optimisation recommendations to improve the current implementation of the MCCP are

- Maximisation of the main clock domain
- Using a quad-lane SRIO link instead of single-lane
- Extension of the code generator to feature creation of optimised program code for state space blocks with an upper triangular A matrix
- Maximisation of DDR3 transfer bandwidth

In future work of the MCCP

- The changes to the design inferred by the new parameter loading interface of the ASIP platform can be implemented. A more scalable parameter loading interface could mean that the parameter sets have to be stored in and/or loaded from, the external memory in a different way.
- A programmable component which packs outputs together to decrease both offloading overhead on the GPP, and packet creation overhead on the MCCP, should be created.
- An optimised offloading scheme could be researched. E.g. offload more `pre_calc` blocks (not only state-space) of a servo application, or conversely maximise the use of large state-space blocks in future servo applications to ensure good performance on a state-space-calculation-optimised MCCP.

As a next step towards the 100 kHz goal, the advantages of using a GPP+FPGA SoC architecture could be investigated. The low-latency communication channels between the

GPP and FPGA with no interference from any other device in such a SoC could be exploited to gain an even larger speedup. For example, a worker could be fully deployed on the SoC, where the GPP handles all non-linear computations, while the FPGA does all of the linear computations. With the higher predictability of the private communication channel, it might be possible to place the FPGA in the critical IO path.

A | List of Abbreviations

AMC	Advanced Mezzanine Card
ASIP	Application Specific Instruction Processor
ATCA	Advanced Telecommunications Computing Architecture
ATMU	Address Translation and Mapping Unit
CARM	Control Architecture Reference Model
DDR	Double Data Rate
DFG	Data Flow Graph
FPGA	Field Programmable Gate Array
GPP	General Purpose Platform
HPPC	High Performance Process Controller
IC	Integrated Circuit
ILA	Integrated Logic Analyser
MCCP	Motion Controller Co-Processor
MIMO	Multi in - Multi out
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
PGEA	Prodrive Gigabit Ethernet AMC
PPA8548	Prodrive Processor AMC 8548
PRIOC QA	Prodrive RapidIO Carrier - Quad AMC
QOR4080	QorIQ P4080 AMC
RAM	Random Access Memory
RIO	RapidIO
SDF	Synchronous Data Flow
SDFG	Synchronous DFG
SRIO	Serial RIO

B | Resource Utilisation

The FPGA on the Xilinx VC707 development kit has a total of

- 303600 LUTs
- 607200 Flip-Flops/Registers
- 1030 36Kb BRAMs (each block can be utilised as two 18 Kb BRAMs)
- 2800 DSP48s

Block	Synthesis	Implementation
srio_gen2_core		
#Slice LUTs (%)	6527 (2.15%)	6376 (2.10%)
#Slice Registers (%)	6963 (1.15%)	6761 (1.11%)
#BRAMs (%)	4.5 (0.44%)	4.5 (0.44%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
srio_gen2_int		
#Slice LUTs (%)	576 (0.19%)	529 (0.17%)
#Slice Registers (%)	851 (0.14%)	755 (0.12%)
#BRAMs (%)	0 (0.00%)	0 (0.00%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
data_distr		
#Slice LUTs (%)	1132 (0.37%)	918 (0.30%)
#Slice Registers (%)	2366 (0.39%)	1759 (0.29%)
#BRAMs (%)	1.5 (0.15%)	1.5 (0.15%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
PSU		
#Slice LUTs (%)	836 (0.28%)	596 (0.196%)
#Slice Registers (%)	587 (0.10%)	487 (0.08%)
#BRAMs (%)	1 (0.10%)	1 (0.10%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
DDR_int		
#Slice LUTs (%)	544 (0.18%)	449 (0.15%)
#Slice Registers (%)	1221 (0.20%)	939 (0.15%)
#BRAMs (%)	0.5 (0.05%)	0.5 (0.05%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
DDR_core		
#Slice LUTs (%)	16149 (5.32%)	15091 (4.97%)
#Slice Registers (%)	12959 (2.13%)	12472 (2.05%)
#BRAMs (%)	1.5 (0.15%)	1.5 (0.15%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
AximCrossbar		
#Slice LUTs (%)	248 (0.08%)	208 (0.07%)
#Slice Registers (%)	510 (0.08%)	293 (0.05%)
#BRAMs (%)	0 (0.00%)	0 (0.00%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
Axim2Avm		
#Slice LUTs (%)	79 (0.03%)	52 (0.02%)

#Slice Registers (%)	100 (0.16%)	100 (0.02%)
#BRAMs (%)	0 (0.00%)	0 (0.00%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
ASIP Platform		
#Slice LUTs (%)	61398 (20.22%)	50761 (16.72%)
#Slice Registers (%)	76983 (12.68%)	67325 (11.09%)
#BRAMs (%)	709 (68.83%)	709 (68.83%)
#DSPs (%)	256 (0.00%)	256 (0.00%)
Additional Logic		
#Slice LUTs (%)	32 (0.01%)	30 (0.01%)
#Slice Registers (%)	33 (0.01%)	33 (0.01%)
#BRAMs (%)	0 (0.00%)	0 (0.00%)
#DSPs (%)	0 (0.00%)	0 (0.00%)
Total		
#Slice LUTs (%)	87521 (28.8%)	75010 (24.71%)
#Slice Registers (%)	102573 (16.89%)	90924 (14.97%)
#BRAMs (%)	718 (69.71%)	718 (69.71%)
#DSPs (%)	256 (9.14%)	256 (9.14%)

Table B.1: MCCP Resource Utilisation Table

References

[1] Ramon R. H. Schiffelers, Wilbert Alberts, and Jeroen P. M. Voeten. “Model-based Specification, Analysis and Synthesis of Servo Controllers for Lithoscanners”. In: *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*. 2012, pp. 55–60. DOI: 10.1145/2508443.2508453.

[2] H. Butler. “Position Control in Lithographic Equipment [Applications of Control]”. In: *Control Systems, IEEE* 31.5 (Oct. 2011), pp. 28–47. DOI: 10.1109/MCS.2011.941882.

[3] A Kalavade and E.A Lee. “The extended partitioning problem: hardware/software mapping and implementation-bin selection”. In: *Rapid System Prototyping, 1995. Proceedings., Sixth IEEE International Workshop on*. June 1995, pp. 12–18. DOI: 10.1109/IWRSP.1995.518565.

[4] Shuai Che et al. “Accelerating Compute-Intensive Applications with GPUs and FPGAs”. In: *Application Specific Processors, 2008. SASP 2008. Symposium on*. June 2008, pp. 101–107. DOI: 10.1109/SASP.2008.4570793.

[5] E.A. Lee and D.G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (Sept. 1987), pp. 1235–1245. DOI: 10.1109/PROC.1987.13876.

[6] RapidIO Trade Association. *RapidIO: The Interconnect Architecture for High Performance Embedded Systems*. Rev. 2.2.

[7] RapidIO Trade Association. *RapidIO Interconnect Specification Part 1: Input/Output Logical Specification*. Rev. 2.2. June 2011.

[8] R.M.W. Frijns et al. “Dataflow-Based Multi-ASIP Platform Approach for Digital Control Applications”. In: *Digital System Design (DSD), 2013 Euromicro Conference on*. Sept. 2013, pp. 811–814. DOI: 10.1109/DSD.2013.126.

[9] Dustin S. Pinedo Hernandez. “A Design-Space Exploration for High-Performance Motion Control”. Master Thesis. Eindhoven University of Technology, Nov. 2012.

[10] M. Bontekoe (ASML). *GID FPGA Based Motion Control Processing Platform*. D000128477-00-GID-001. June 2014.

[11] IEEE. “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[12] Xilinx. *VC707 Evaluation Board for the Virtex-7 FPGA*. UG885 (v1.3). Aug. 2013.

[13] Xilinx. *LogiCORE IP Serial RapidIO Gen2 Endpoint v3.1*. PG007. Nov. 2013.

[14] Xilinx. *Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v2.1*. UG586. June 2014.

[15] Xilinx. *LogiCORE IP Integrated Logic Analyzer v4.0*. PG172. Apr. 2014.