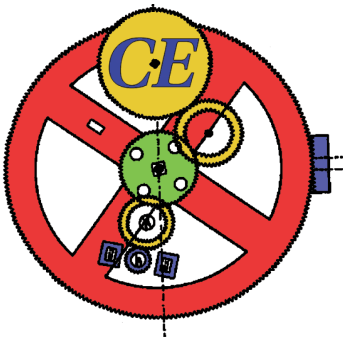

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

Performance improvement of motion-control applications using multi-ASIP in FPGA

Sumedh W. Jambekar

CE-MS-2014-12

Abstract

ASML is a world leading supplier of complex lithography machines for the semiconductor industry. A lithography machine consists of many subsystems, e.g., *Light Source*, *Lens*, *Reticle handler*, *Reticle stage*, *Wafer handler* and *Wafer stage*, which synchronize together to make the machine work. The *Reticle stage* holds the circuit pattern, also known as reticle and the *Wafer stage* module holds the wafer. The UV light from the light source is projected on the circuit pattern, which is then passed through the lens to imprint the pattern on the wafer. Since the circuit pattern has to be imprinted on the wafer, the movement of the modules; *Reticle stage* and *Wafer stage* should be synchronized in six degrees of freedom (DoF) with nanometer accuracy. To employ the movement of the subsystems, motion controllers are used in ASML, and Long Stroke and Short Stroke controllers are responsible for the movement of a part of the *Wafer stage* subsystem. It has been envisioned that future lithography machines, because of its high precision mechatronic requirements, will need motion control algorithms, that run at higher sampling frequencies with a severely reduced IO latency budget. Current hardware architectures will not be able to meet the demands of these future motion control algorithms. In this thesis, we propose an architecture, that uses a multi-ASIP in FPGA as an accelerator in conjunction with a CPU, which acts as a master to run the motion control applications. The proposal of using multi-ASIP FPGA in conjunction with CPU is based on the analysis carried out previously in ASML. It was observed that a sampling frequency exceeding 100 KHz can be obtained after deploying the Long Stroke controller and Short Stroke controller on a multi-ASIP platform in FPGA. However, this work considered only the data flow and not the supervisory control. After carrying out detailed analysis, we could predict that a sampling frequency of 40 KHz could be achieved by offloading the compute intensive blocks present in the Long Stroke and Short Stroke controller from the CPU to FPGA. The sampling frequency of 40 KHz can be achieved by considering, both the data flow and supervisory control, and the communication between the CPU and FPGA. Finally, after offloading the compute intensive blocks from the CPU on the multi-ASIP FPGA, and after implementing the data flow and supervisory control and communication mechanism between the CPU and FPGA, we can justify that the sampling frequency of 40 KHz can be achieved.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Performance improvement of motion-control applications using multi-ASIP in FPGA

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Sumedh W. Jambekar
born in Solapur, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Performance improvement of motion-control applications using multi-ASIP in FPGA

by Sumedh W. Jambekar

Abstract

ASML is a world leading supplier of complex lithography machines for the semiconductor industry. A lithography machine consists of many subsystems, e.g., *Light Source*, *Lens*, *Reticle handler*, *Reticle stage*, *Wafer handler* and *Wafer stage*, which synchronize together to make the machine work. The *Reticle stage* holds the circuit pattern, also known as reticle and the *Wafer stage* module holds the wafer. The UV light from the light source is projected on the circuit pattern, which is then passed through the lens to imprint the pattern on the wafer. Since the circuit pattern has to be imprinted on the wafer, the movement of the modules; *Reticle stage* and *Wafer stage* should be synchronized in six degrees of freedom (DoF) with nanometer accuracy. To employ the movement of the subsystems, motion controllers are used in ASML, and Long Stroke and Short Stroke controllers are responsible for the movement of a part of the *Wafer stage* subsystem. It has been envisioned that future lithography machines, because of its high precision mechatronic requirements, will need motion control algorithms, that run at higher sampling frequencies with a severely reduced IO latency budget. Current hardware architectures will not be able to meet the demands of these future motion control algorithms. In this thesis, we propose an architecture, that uses a multi-ASIP in FPGA as an accelerator in conjunction with a CPU, which acts as a master to run the motion control applications. The proposal of using multi-ASIP FPGA in conjunction with CPU is based on the analysis carried out previously in ASML. It was observed that a sampling frequency exceeding 100 KHz can be obtained after deploying the Long Stroke controller and Short Stroke controller on a multi-ASIP platform in FPGA. However, this work considered only the data flow and not the supervisory control. After carrying out detailed analysis, we could predict that a sampling frequency of 40 KHz could be achieved by offloading the compute intensive blocks present in the Long Stroke and Short Stroke controller from the CPU to FPGA. The sampling frequency of 40 KHz can be achieved by considering, both the data flow and supervisory control, and the communication between the CPU and FPGA. Finally, after offloading the compute intensive blocks from the CPU on the multi-ASIP FPGA, and after implementing the data flow and supervisory control and communication mechanism between the CPU and FPGA, we can justify that the sampling frequency of 40 KHz can be achieved.

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-12

Committee Members :

Advisor: dr. ir. Zaid Al-Ars, CE, TU Delft

Chairperson: prof. dr. Koen Bertels, CE, TU Delft

Member:	ir. Nikola Gidalov, MC&I, ASML
Member:	dr. ir. Gerard Janssen, WMC, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ASIP Application-specific instruction-set processor

FPGA Field-programmable gate array

CARM Control architecture reference model

DSL Domain specific language

IC Integrated circuit

GPU Graphics processing unit

ASIC Application-specific integrated circuit

GPP General-purpose processor

VPU Vector processing unit

SPU Scalar processing unit

LoS Long stroke

SS Short stroke

sRIO serial Rapid IO

DDR3 Double data rate type three

SDRAM Synchronous dynamic random access memory

IO Input/Output

DoF Degree of Freedom

SoC System on Chip

RPC Remote procedure call

SI Sensor interface

MS Measurement system

AS Actuator system

AMC Advanced mezannine card

HPPC High performance process control

IDT Input data terminal

IJT Input inject terminal

IET Input event terminal

ODT Output data terminal

OET Output event terminal

ATCA Advanced telecommunications computing architecture

Acknowledgements

You never make it to the destination alone, it is the people surrounding you that helps you reach the destination. I want to thank all those people, who worked with me and helped me in completing my project. First of all, I am extremely grateful to ASML for providing me the opportunity to carry out my graduation project. I had a wonderful time working here, that helped me improve my social, professional and technical skills. I am also thankful to Prodrive for providing the infrastructure to carry out all the experiments required for my project. I also thank Luke Lemmen, Eric van Uden, Mels van Broekhoven and Jerry Jacobs for supporting me in understanding the test set-up.

I am also thankful to all my supervisors, both in the university and at ASML, for showing confidence in me and guiding me throughout my project. First of all, I am extremely grateful to my university supervisor, Zaid Al-Ars for providing me the opportunity to carry out the project under his supervision. I thank him for being patient in reviewing my report thoroughly time and again, and providing me the appropriate feedback. The feedback provided was really helpful, as it helped me improve my technical, analytical and writing skills. I would also like to thank my committee member Gerrard Janssen for readily accepting to be a part of my thesis committee.

I am extremely grateful towards all my ASML supervisors, for believing in me and supporting me in every step of the project. First of all, I am thankful to Jeroen Voeten and Ramon Schiffelers, for allowing me to work on the project of my choice. It was an excellent experience working on this project, and also working under their supervision. I also thank them for being patient in answering all my questions, I really appreciate that. My special thanks to Ramon for providing me all the resources required to carry out my project smoothly. Without his support, it was impossible to finish the project.

I would also like to express my sincere thanks to Nikola Gidalov and Marcel Bontekoe for their support. I thank Nikola for making me acquainted with the working of the control applications here at ASML. I am also thankful to him for providing me technical feedback and feedback on the measurements and results obtained in the project. I would like to express my thanks to Marcel, for providing me the required knowledge about the hardware, and being patient in answering all the queries about FPGA. I am also deeply grateful towards my daily supervisor Raymond Frijns for supporting me and helping me in every step of the project. I sincerely thank him for being patient with me in answering all my queries, even in the weekends and in the nights, I really appreciate this.

My special thanks to my friends Shreya, Santhosh, Aditya Deshpande, Anupam Chahar, Kaushal Butala, Mani Kaustubh, Adithya Pulli, Arun, Abhimanyu, Harshitha, Sriram, Phani Kiran, Visweswaran and Manjunath for supporting me and being with me for all these two years in Netherlands.

Finally, I would like to thank my parents, my sister and all my relatives and friends, for supporting me emotionally throughout my master's. It would not have been possible to finish my master's without their support.

Sumedh W. Jambekar
Delft, The Netherlands
August 20, 2014

1 Introduction

ASML is a world leading supplier of complex lithography machines for the semi-conductor industry. The lithography machines in ASML are photo-lithography machines which use light as a source to imprint a circuit pattern on a wafer. Creation of an integrated circuit (IC) involves many steps, and photo-lithography is one of the most important steps involved. The photo-lithography machines consist of many subsystems, e.g., *light source*, *reticle handler*, *lens*, *wafer handler* and *wafer stage*. A reticle is a circuit pattern that is to be imprinted on a wafer, and is placed between the light source and the lens. A wafer, on which a circuit pattern is to be imprinted, is covered by a thin layer of photo-resist material. In order to imprint a circuit pattern on a wafer, the UV light from the light source is projected on the retilcle (circuit pattern), which is then passed through the lens, and is finally projected on a photo-resist material present on a wafer. The UV light reacts with a photo-resist material and imprints the circuit pattern on a wafer. The photo-lithography step is repeated many times during the creation of integrate circuit (IC), where in each of the steps, a layer of the circuit pattern is imprinted on top of the previous layer. Figure 1.1 shows the photo-lithography process, where a circuit pattern is imprinted on a wafer by exposing it to the UV light. This entire photo-lithography process can be realized by either moving the light source, or by moving the reticle and wafer stages. Since the effort required in moving the light source is costly, it is kept constant, and the reticle and wafer stages are synchronized to imprint a circuit pattern on a wafer. Since, each of the layers on a wafer must be synchronized with the previous layers, the wafer and reticle stages should be moved in six degrees of freedom (DoF) with nanometer accuracy. The movement of the subsystems in six degrees of freedom and the desired nanometer accuracy is satisfied by employing the motion controllers. The motion controllers, accept input from the sensors, processes these inputs to minimize the errors and actuate the susbsystem (plant) accordingly. The Long Stroke controller and Short Stroke controller are the controllers that are used in the movement of the Wafer Stage subsystem present in the lithography machine. The controllers receive the input periodically, calculate the output and send the output to the actuators. Sampling frequency can be defined as the rate at which the new samples arrive and IO-delay can be defined as the delay between sensing and actuating.

1.1 Problem Description

The motion control applications in ASML are currently running on the general purpose processors (GPPs), as shown in the Figure 1.2. The figure shows the control blocks (CB), accepting the inputs from the sensors (S), processing these inputs and sending the outputs to the actuators (A). Figure 1.2 also introduces Supervisory Control and an ATCA rack. From the Figure 1.2, it can be observed that the control blocks are mapped

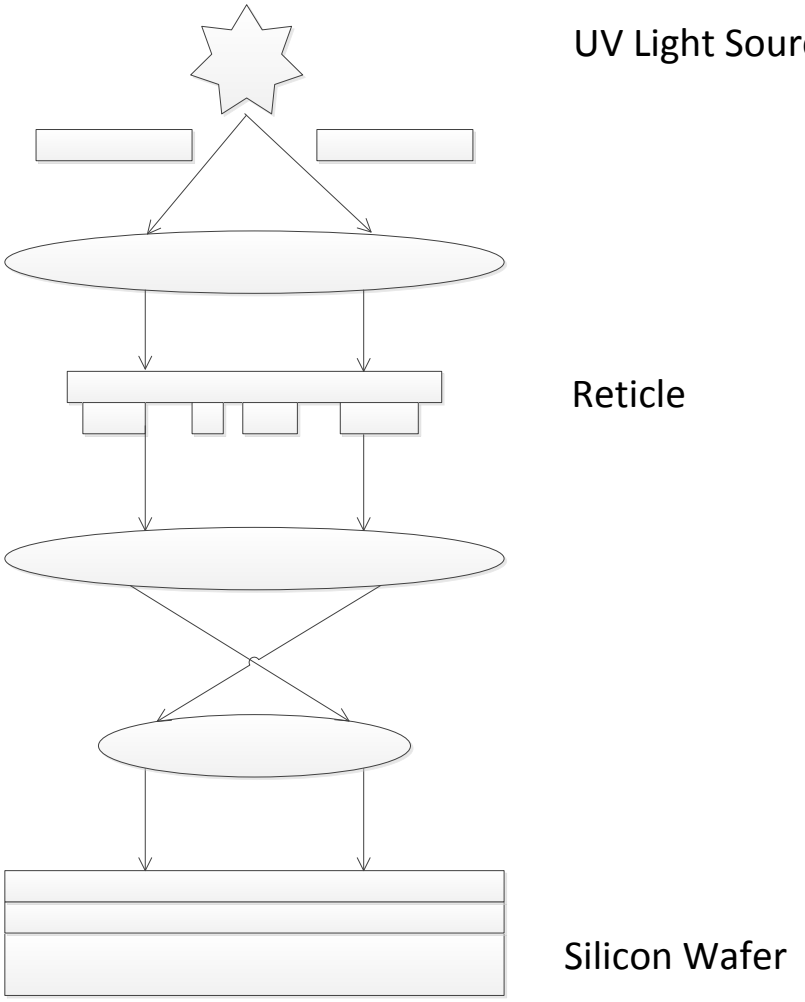

Figure 1.1: Exposure of a UV light on a wafer

on an ATCA rack, that consist of Host and the HPPCs. HPPC is an abstraction of a processor. Host is responsible for communicating with the HPPCs during initialization and run-time. Host is connected to the HPPCs via ethernet, and the HPPCs are connected to one another via serial Rapid IO connectivity. The HPPCs are general-purpose processors which run the control blocks. The maximum sampling frequency at which the Long Stroke and the Short Stroke controller can be executed on the general purpose processors is 20 KHz. It has been predicted that future lithography machines, because of their high mechatronic requirements, will require motion control algorithms which are complex [5]. It has also been envisioned that the architectures currently employed by ASML would no longer be able to meet the requirements of the future motion control applications. To meet the demands of the future motion control applications, high performance architectures, which can run the motion control applications at a higher sampler rate and with the reduced IO delay are required. There are many such architectures that can be employed to meet the demands of the future motion control applications.

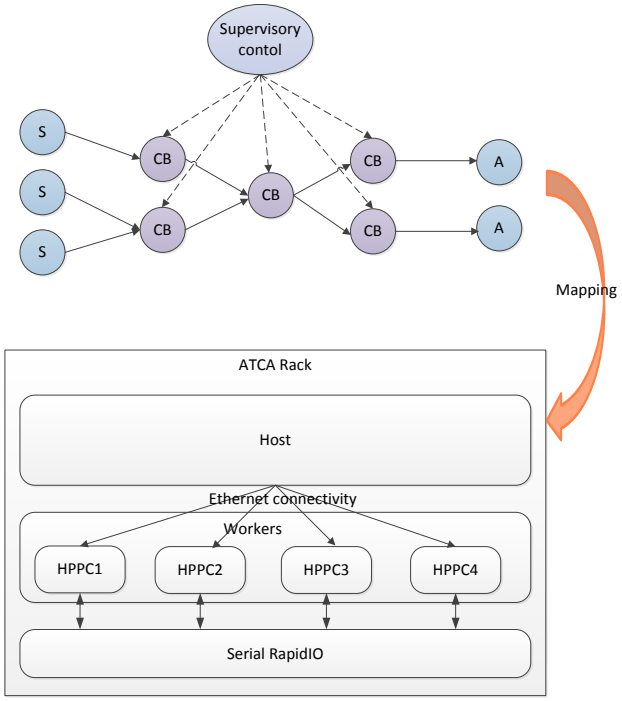

Figure 1.2: Motion control application deployed on the general purpose processors run at a sampling frequency up to 20 KHz

Figures 1.3 [1] and 1.4 [1] shows the benchmark application; Long Stroke (LoS) and the Short Stroke (SS) controllers, that is envisaged to be used in the future. The benchmark application was developed in close collaboration with the ASML mechatronics research group. The Long Stroke and the Short Stroke controllers are responsible for actuating the Wafer Stage in six degrees of freedom with a nanometer accuracy.

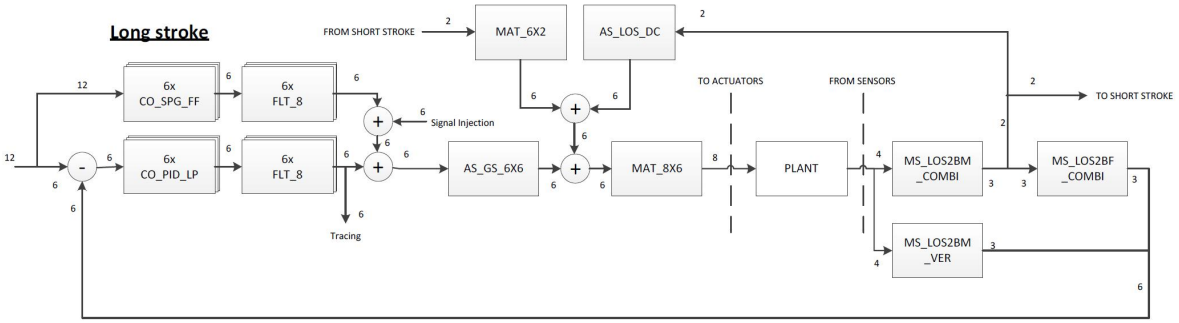

Figure 1.3: Long Stroke controller [1]

The Long Stroke controller provides a coarse positioning of the Wafer Stage with a micrometer accuracy, whereas, a Short Stroke controller fine tunes the Wafer Stage position with a nanometer accuracy. Since a Short Stroke controller positions a Wafer

Stage with a nanometer accuracy, the information processed by a Short Stroke controller is more when compared to the Long Stroke controller. Hence, it can be seen from Figure 1.4 that, the Short Stroke controller consist of large state-space blocks, which have 220 states, 11 inputs and 11 outputs. The Short Stroke controller that is currently present in the machine, do not contain the state-space blocks, and the Long Stroke controller currently present in the machine, is more or less the same as that present in the benchmark application.

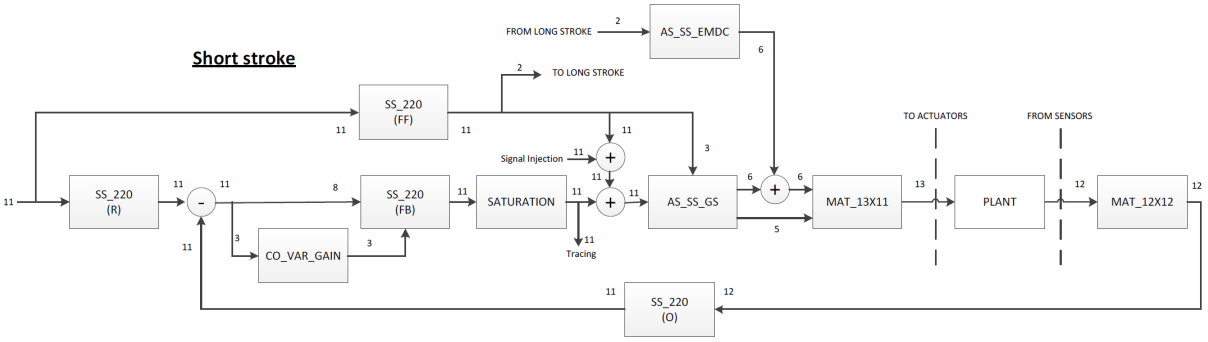

Figure 1.4: Short Stroke controller [1]

Previous work [5] has shown that, it is possible to run the motion control applications at high sample frequencies and small IO-delays on an FPGA. In this work, the benchmark application was deployed on a heterogeneous platform in FPGA. It was observed from this previous work that using FPGA as a hardware platform provided a good trade-off between the flexibility and performance, and a sampling frequency of 133 KHz was achieved. However, this sampling frequency was achieved by considering only the data flow and supervisory control was not taken into consideration. Figure 1.5 shows the benchmark application deployed on an FPGA. It can be observed from the figure that only the data flow was taken into consideration. It can also be seen that there was no communication bottleneck as CPU was not considered and the benhcmark application was deployed only on the FPGA.

Deploying the entire benchmark application on an FPGA, considering both the data flow and the supervisory control, as shown in the Figure 1.6, is too big a step to achieve, and involves, lot of time and effort. Moreover, since the architecture currently used in the software framework in ASML differs completely with the FPGA architecture, it is difficult to predict whether the FPGA arachitecture fits in the current software framework.

Since deploying the complete benchmark application on an FPGA, including the data flow and the supervisory control, is too big a step, in this project, we employ an intermediate step, where, FPGA would be used as an accelerator in conjunction with the general purpose processor, as shown in Figure 1.7. In our work, we deploy the compute intensive blocks present in the benchmark application from CPU on the FPGA, and we also take supervisory control into consideration. Despite the communication between CPU and FPGA, and despite considering the supervisory control, we envision that a

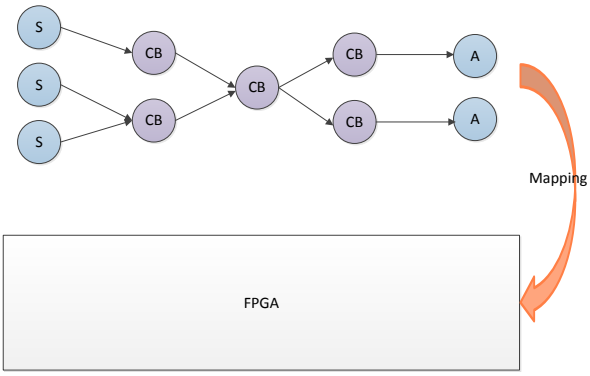

Figure 1.5: Motion control application deployed on an FPGA run at a sampling frequency of 100 KHz

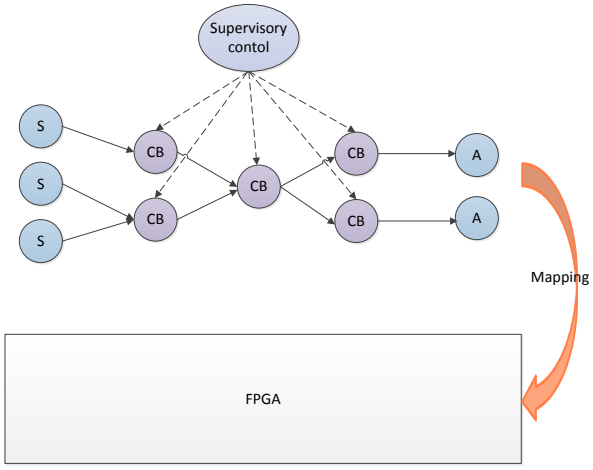

Figure 1.6: Motion control application deployed on an FPGA including both the data flow and the supervisory control

sampling frequency of 40 KHz could be achieved after using this intermediate step.

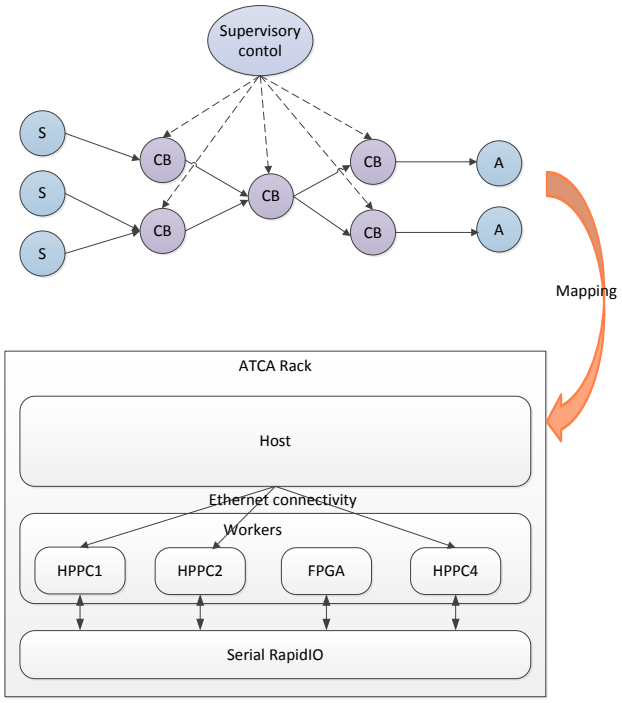

Figure 1.7: FPGA used as an accelerator in conjunction with general purpose processor

1.2 Project Goal

The goal of the project is to:

1. Investigate whether a multi-ASIP FPGA can be integrated within the current software framework.
2. Investigate whether using multi-ASIP FPGA in conjunction with GPP increases the sampling frequency of the benchmark application.

1.3 Approach

In the initial phase of the project, a detailed study was carried out to thoroughly understand the control architecture reference model (CARM), various terminologies used in the CARM model, software architecture and the hardware architecture. In the next phase, using the CARM tool chain, the execution times of all the blocks present in the Short Stroke controller and Long Stroke controller were obtained. A block which has the highest execution time and from which the data-level parallelism could be exploited was chosen for deployment on a multi-ASIP FPGA. An analysis was then carried out on the chosen block, and it was predicted that the sampling frequency of the benchmark application can be increased to 40 KHz. In the next phase, the hardware architecture was studied in detail. Previous work carried out at ASML on the FPGAs was studied thoroughly. An analysis was then carried out on different hardware architectures, and based on the analysis, a hardware architecture that could best fit for our project was chosen. In the final phase, the implementation increments were defined in detail.

1.4 Research Hypothesis

The sampling frequency of the benchmark application can be increased to 40 KHz by offloading the compute intensive blocks from GPP to FPGA. The sampling frequency of 40 KHz can be achieved by considering the communication between GPP and FPGA and also considering the supervisory control.

Following are the research questions that are to be answered in the thesis:

1. Which part of the motion controller should be deployed on an FPGA?
2. Which part of the software architecture should be modified?
3. What are the different hardware architectures that are feasible?

1.5 Report Organization

This report is organized as follows: Chapter 2 provides the background and the terminologies used in CARM in detail. Chapter 3 explains in detail the benchmark application used in the project. Chapter 4 discusses in detail the previous work carried out in ASML. Chapters 5 and 6 describe in detail the software analysis and architecture and hardware

analysis and architecture respectively. In chapter 7 the steps carried out in implementation are described in detail. Chapater 8 discusses the measurements and the results and finally, in chapter 9, the conclusion and the future work are presented.

CARM background information

2

In this chapter, a detailed description is provided about the various terminologies used in the current software framework. A detailed description of the current software and the hardware architecture is also presented. All the information provided in this chapter would be a foundation for the rest of the chapters.

2.1 Control Architecture Reference Model (CARM)

The controllers used in ASML are modeled using Control Architecture Reference Model (CARM). Time and again it has been proven that the model based approach is efficient and advantageous. The model based approach identifies the errors made in the design much earlier in the process and helps in saving both time and money [2]. The CARM model is divided into 3 layers: Application layer, Mapping layer and Platform layer. The Application layer describes the behaviour and the control logic, the Platform layer describes the hardware such as processors, IO boards and so on and Mapping layer describes the mapping of the control blocks on the platform [6]. Figure 2.1 [2] shows the different CARM domain specific languages (DSLs) and their positioning in the Y-chart paradigm.

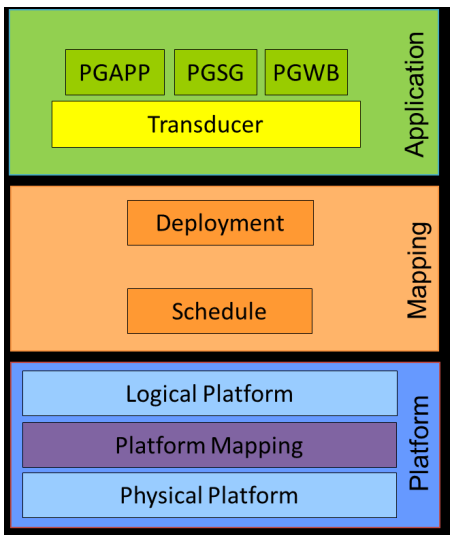

Figure 2.1: CARM Domain Specific Languages [2]

2.2 Modeling controllers in CARM

In this section, we describe in detail, the different terminologies used in CARM.

2.2.1 Control loop structure

Controllers are employed to control the behaviour of the system, generally known as plant. A typical control loop, as shown in the Figure 2.2, consist of a plant that is to be controlled, a controller that controls the plant, sensors and actuators.

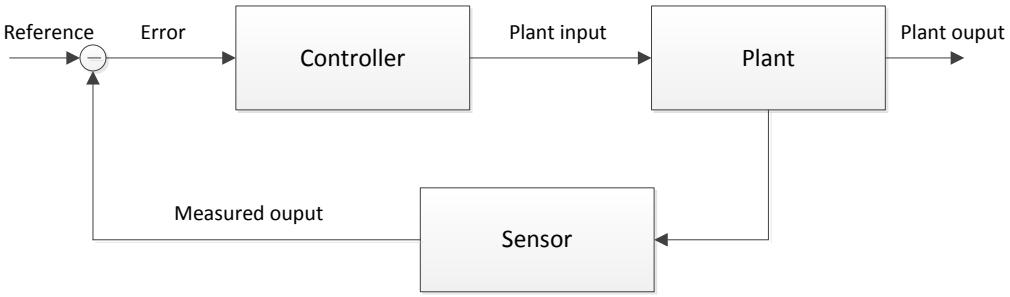

Figure 2.2: Diagram showing the control loop

A reference is the desired signal that a system (plant) must follow, and the measured output is the position of the plant sensed by the sensor. The difference between the reference and the measured output is calculated, and is given as an error to the controller. The controller tries to minimize this error signal and actuates the plant accordingly. Since the next state of the plant depends on the previous state, this kind of control loop is also known as the feedback control or servo control. The Long Stroke controller and Short Stroke controllers, explained previously, are examples of a closed loop control.

2.2.2 ServoGroup

In CARM, controllers are modelled by so called ServoGroups. A ServoGroup [7], which is a closed loop, is a collection of control blocks, that is responsible for actuating a part of a sub-system. There are many ServoGroups present in the system which work together to actuate a physical entity such as the wafer stage. Every ServoGroup has a sampling frequency and all the WorkerBlocks present in the ServoGroup have the same sampling frequency as that of a ServoGroup.

Figure 2.3 shows the graphical representation of a ServoGroup. A typical ServoGroup consists of a sensor interface (SI), a measurement system block (MS), control blocks (WB1, WB2 and WB3), an actuator system (AS) and the motor interface blocks (MI1 and MI2).

2.2.2.1 WorkerBlock

A WorkerBlock [7] models a control block, and is a smallest entity that can be accessed in the ServoGroup. The examples of WorkerBlocks are filter block, matrix block, gain block, summation block. A WorkerBlock performs the calculations, based on the inputs

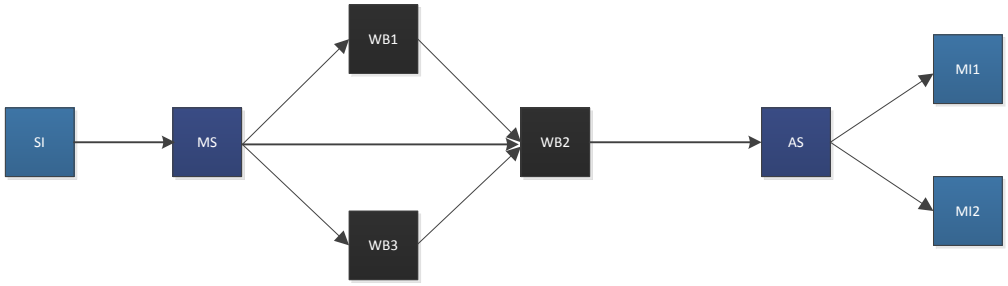

Figure 2.3: Graphical representation of a ServoGroup

and the parameter sets, and produces the output. The WorkerBlocks are connected to one another via data ports to form the ServoGroup. Figure 2.4 shows the graphical representation of a WorkerBlock. Below are some of the concepts that are associated with a WorkerBlock.

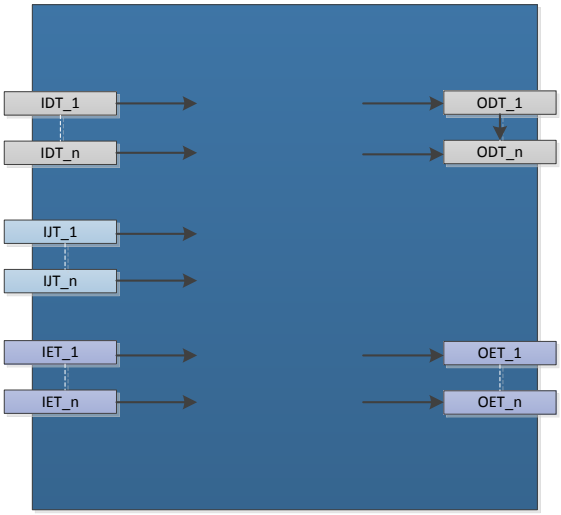

Figure 2.4: Graphical representation of a WorkerBlock

Property: Every WorkerBlock has properties [7] which are fixed at design time. The properties of a WorkerBlock are defined in the network definition file. The number of parameter sets a WorkerBlock can have is one of the examples of a WorkerBlock property.

Parameters and Parameter Sets: Apart from the input data, the output of a WorkerBlock is also dependent on its parameters [7]. A parameter set is a collection of all the parameters of a block, whose initial value is known at design time. A WorkerBlock can change its active parameter set at run-time. The number of parameter sets a WorkerBlock can have is fixed and is known at design time. The format of a parameter set is shown in the Figure 2.5.

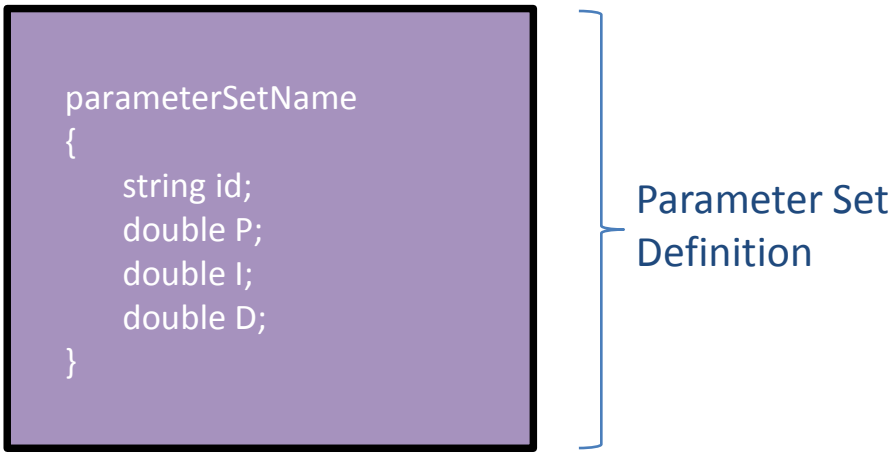

Figure 2.5: Graphical representation of a parameter set

An example of 3 parameter sets; "PS1", "PS2" and "PS3" are shown in the Figure 2.6.

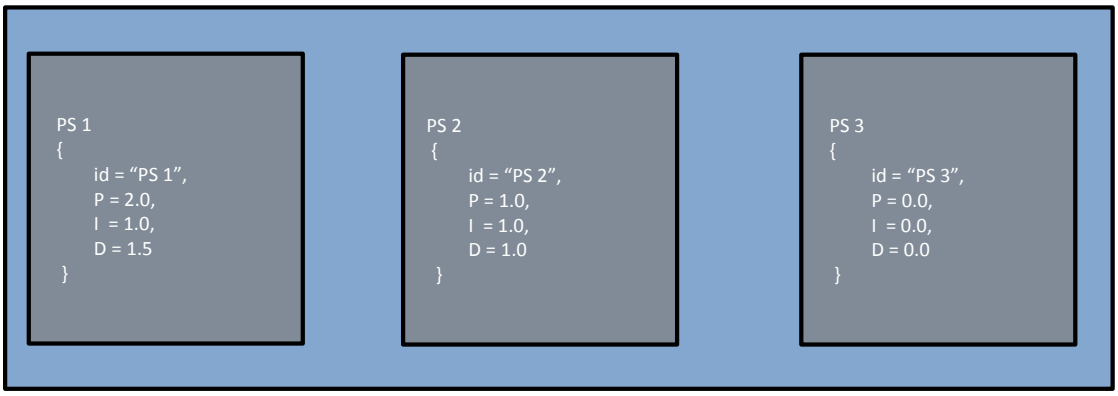

Figure 2.6: Figure showing the parameter sets of a WorkerBlock

Terminals: A WorkerBlock consists of 3 types of input terminals and 2 types of output terminals [7]. The types of input terminals of a WorkerBlock are input data terminal, input inject terminal and input event terminal. The types of output terminals of a WorkerBlock are output data terminal and output event terminal. A WorkerBlock receives the input data and input inject data every sample and produces output data every sample. The input event data is received only when a Control Mode switch has to be performed. The result is the change in the Control Mode. WorkerBlocks receive their input data from other WorkerBlocks on which they are dependent. The input data is received by a control block from the blocks in a ServoGroup which it is dependent on. The input inject data received by a WorkerBlocks is used for diagnostic purposes.

Calculations of a WorkerBlock: A WorkerBlock has a function fullCalc() that performs the calculations for that WorkerBlock. In order to minimize the IO delay (delay between the sensors and actuators), a fullCalc() function can be divided into preCalc() function and postCalc() function [7]. The preCalc() is invoked in the non-time critical part of the sample and the postCalc() is invoked in the time-critical part of the sample. Whether a WorkerBlock should operate in the time-critical or nontime-critical part of the sample, is a property of a WorkerBlock, and can be configured at design time. Most of the WorkerBlocks present in the servo-group fall into the category of the time-critical section, and the WorkerBlocks such as error-checker and so on, fall into the category of non-time-critical section.

Time critical part of the sample: In order to minimize the IO delay between the sensors and the actuators, the calculations of the blocks might be divided into postCalc() and preCalc() functions. The time critical part of the sample consists of all the postCalc() functions of the blocks, which are needed to compute the output sample of the servogroup. The time critical part of the sample might also consist of the fullCalc() function if it is not further divided into postCalc() and preCalc() functionalities. Figure 2.7 shows the time critical part of the sample.

Non-time critical part of the sample: The non-time critical part of the sample consist of all WorkerBlock functionality that is not time-critical. Figure 2.7 describes the non-critical part of the sample.

WorkerBlock state: Every WorkerBlock can be in one of the two states: *On* or *Off* [7]. In an *On* state, the WorkerBlock performs its calculations and produces its output. In an *Off* state, the output of a WorkerBlock is either zero or is same as the input of a WorkerBlock. The block state concept is used to set a controller in an open loop or closed loop.

2.2.2.2 Schedule

A schedule is an execution order of blocks on a Worker. Since the schedule is static, the execution order of the blocks is known at compile time and does not change at runtime. Figure 2.7 shows the graphical representation of a schedule. Apart from the blocks that perform the calculations, a schedule also consist of a control mode switch block, a ServoGroup Queue block and the blocks which perform the background activities.

2.2.2.3 Control Mode

A Control Mode [7] is a specific combination of a parameter set of every WorkerBlock present in a ServoGroup. A Control Mode selects for each WorkerBlock of a ServoGroup, the active parameter set. A ServoGroup behavior can be changed by switching from one ControlMode to another. Figures 2.8 and 2.9 show five WorkerBlocks and two control modes, where each of the control modes have a specific parameter set of every WorkerBlock.

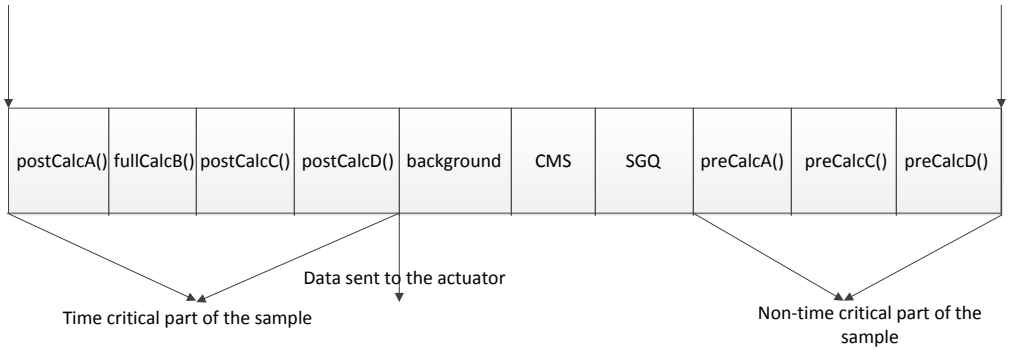

Figure 2.7: A schedule showing the execution order of the blocks

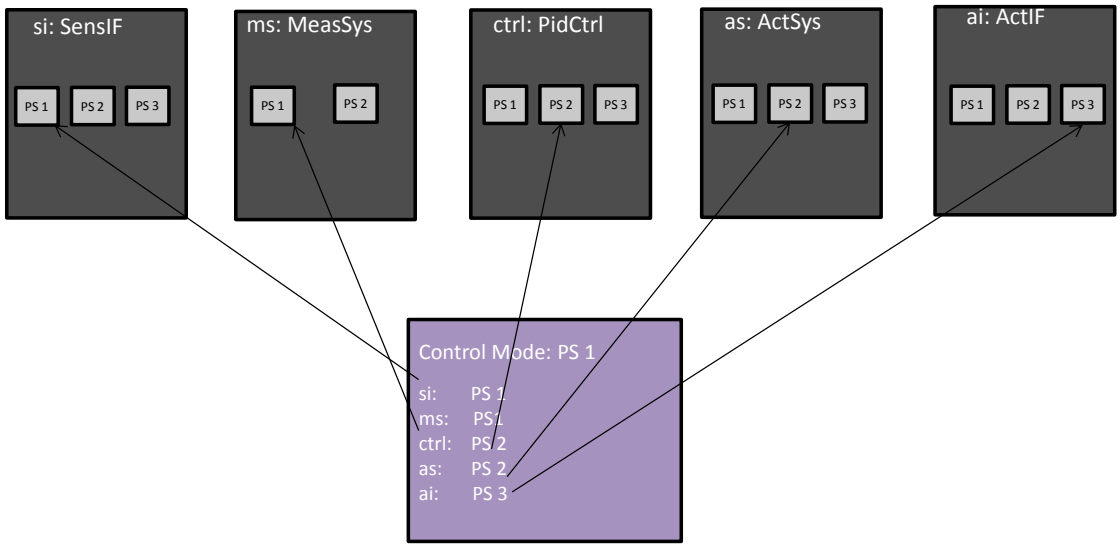

Figure 2.8: Graphical representation of control mode one

2.2.2.4 BlockGroup

A BlockGroup [7] is a subset of the WorkerBlocks present in a ServoGroup on which the state behaviour can be performed. Figure 2.10 shows a ServoGroup consisting of multiple BlockGroups.

2.3 Modeling execution platforms in CARM

This section describes the physical platform, logical platform and platform mapping domain specific languages in detail.

2.3.1 Physical platform

The physical platform domain specific language describes in detail the underlying platform present in the lithoscanner. The concepts involved in the physical platform language

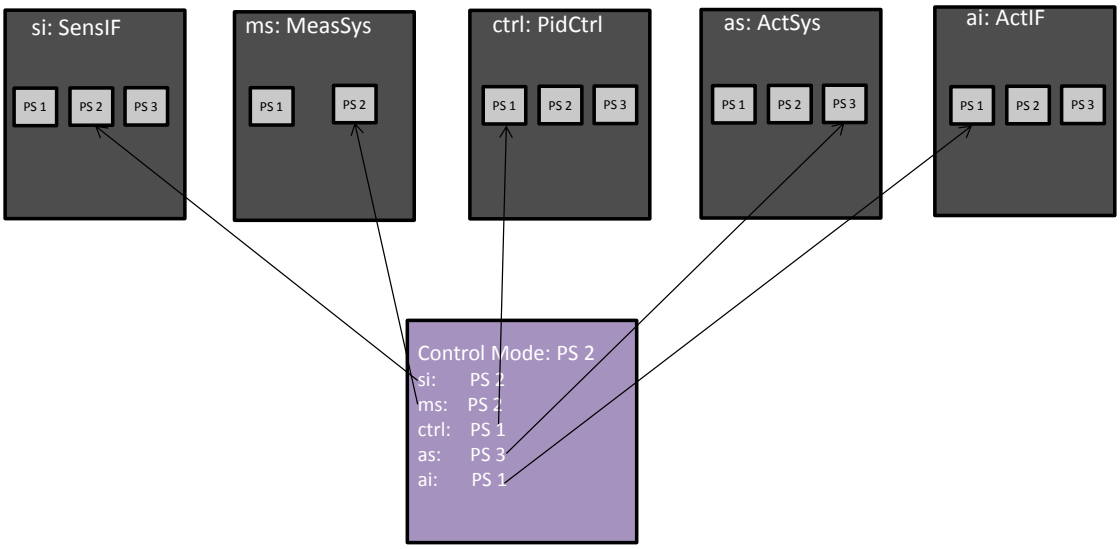

Figure 2.9: Graphical representation of control mode two

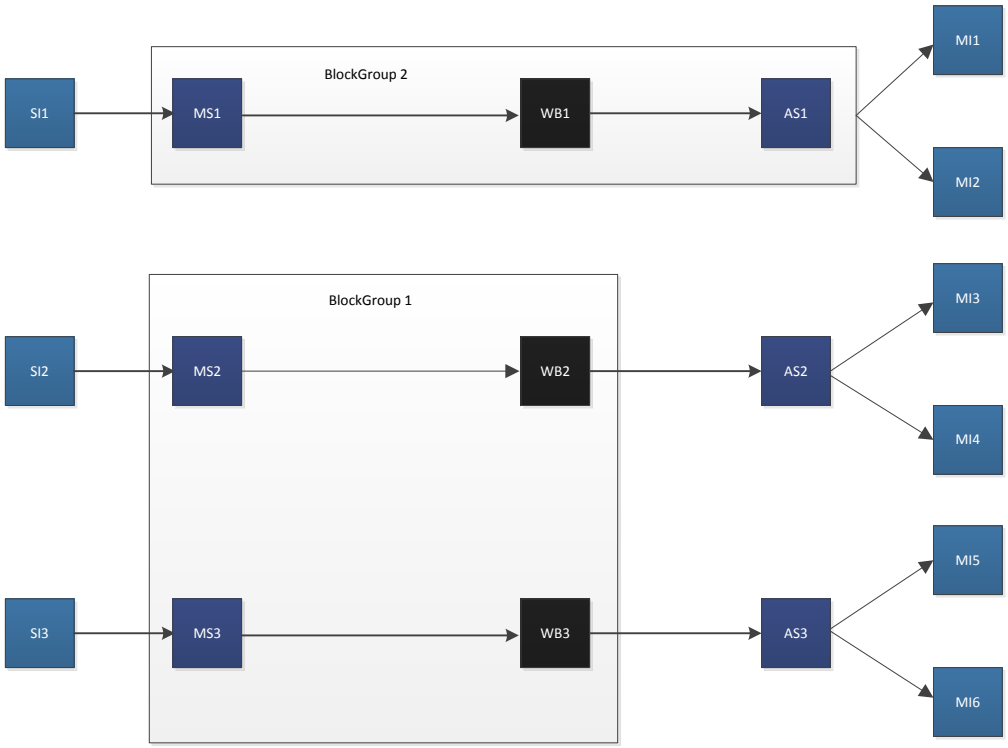

Figure 2.10: Figure showing a ServoGroup consisting of multiple BlockGroups

are the HPPCs (an abstraction of the processor), transducers, switches, ATCA racks and so on [2]. The physical platform language does not describe the configuration information of the platform, the configuration data of the platform is present in the platform mapping language.

2.3.2 Logical platform

The logical platform language is an abstraction of the physical platform language [2]. The concepts involved in the logical platform language are Worker (an abstraction of the HPPCs), processing unit (an abstraction of the processor/core), channels (an abstraction of the connectivity) and so on.

2.3.3 Platform mapping language

The platform mapping DSL describes the mapping of the logical platform language on the physical platform language [2]. The platform mapping language consist of the configuration data, such as the sampling frequency. The platform mapping language also consist of the mapping information of the logical platform language to physical platform language. Some of the examples of platform mapping are the mapping of Workers to HPPCs and channels to network connectivity.

2.4 Controller mapping

As explained earlier, the application consists of the entire servo controllers, which are described by the application DSLs, and the platform consists of the underlying hardware of the lithoscanners, which are desribed by the platform DSLs. The mapping of the servo controllers on the execution platform are described using the mapping DSLs[2]. Figure 2.11 shows an example of the mapping of a single control application on an execution platform. It can be seen from the figure that, the WorkerBlocks are mapped on the Workers and the WorkerBlock communication is mapped on the network. The lithography machine consists of hundreds of sensors, actuators and many ServoGroups, which have to be deployed on the execution platform.

The mapping language also consist of the information about which control block should be deployed on which processor. Since a processor executes more than one control block, the processor should know in advance, the order of execution of the control blocks. The schedule DSL is responsible for generating the schedule, according to which the processor executes the blocks [2]. As a first step, the schedule DSL extracts the information about the applications from the application DSLs and the deployment information from the deployment DSL. Out of the information obtained from the application and deployment DSLs, a block dependency graph is created, as shown in the Figure 2.12 [2]. This is done using model-model transformation. A block dependency graph consists of the control blocks and the dependencies between them. The information present in the block dependency graph is used to generate the schedule of the processors, as shown in the Figure 2.12 [2].

2.5 Runtime reconfiguration

In this section, the concepts; control mode switch and Host, are described in detail.

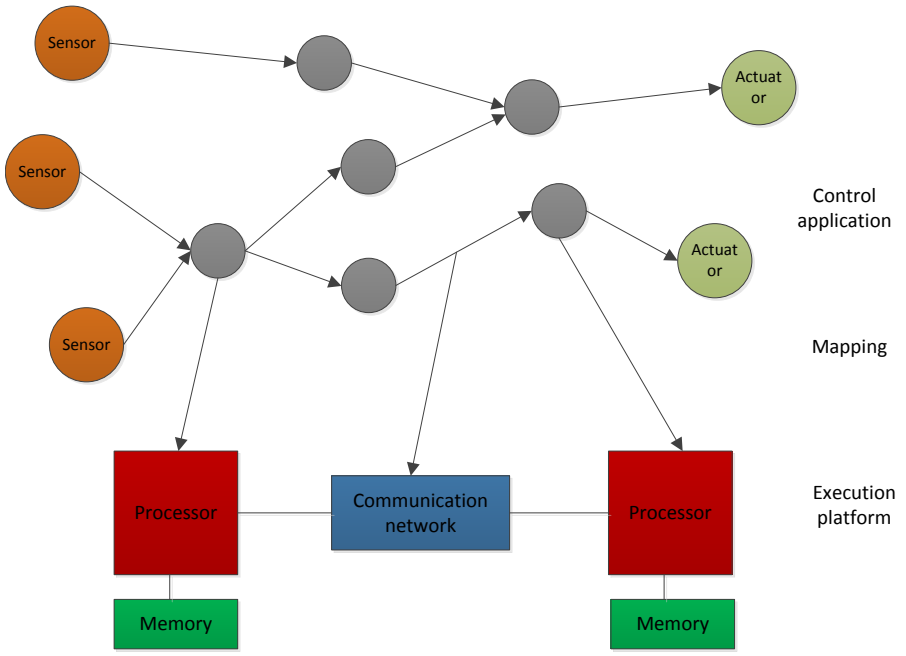

Figure 2.11: Controller mapping on the execution platform

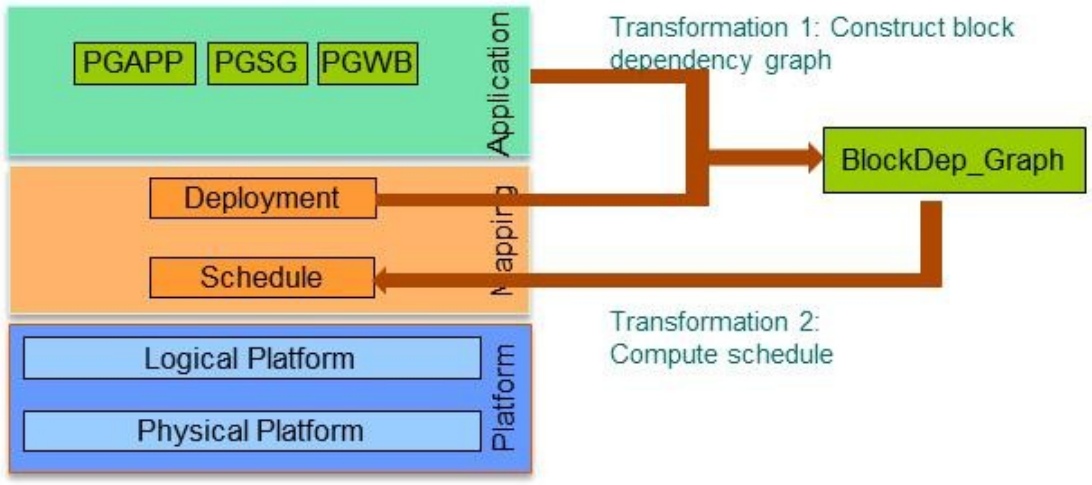

Figure 2.12: Block dependency graph creation [2]

2.5.1 Supervisory Control

Supervisory control is responsible for performing the state-change behaviour and Control Mode Switches on WorkerBlocks. Host is an entity that is responsible for communication with the Workers, both during the initialization and run-time. Host is connected to the Workers via ethernet, whereas, the Workers are connected to each other via serial rapidIO. Figure 2.13 shows the graphical representation of the Host. During initialization, a Host is responsible for deploying the WorkerBlocks present in a ServoGroup on

a Worker and connecting them, enforcing the execution order on the WorkerBlocks according to the static schedule generated, loading the Parameter Sets in the non-cachable memory and performing a Control Mode Switch to choose the initial parameter set on which a WorkerBlock must operate. At run-time, the Host communicates with the Workers to communicate the run-time change in the value of parameter sets, performing the state-change behaviour and performing a Control Mode Switch.

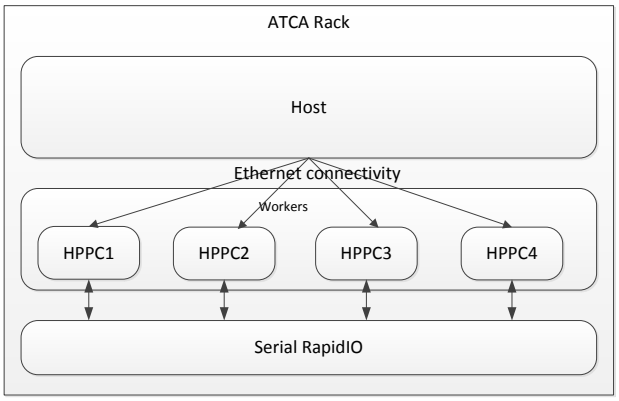

Figure 2.13: Figure showing a Host and the Workers in an ATCA rack

2.5.2 Control Mode Switch

Control Mode Switch is performed by the Host when there is a need to operate a physical entity (plant) in a different mode. As mentioned earlier, a Control Mode is a specific selection of active parameter sets of all WorkerBlocks present in a ServoGroup. As soon as the Control Mode Switch is signalled, the Host communicates this information to a ServoGroup present on the Worker. As a result, all the parameter sets that constitute the Control Mode that should be made active, are realized by fetching the all memory parameter values from non-cachable memory to cachable memory. Once all the parameter sets are fetched, a Control Mode Switch is performed to activate the Control Mode.

2.5.3 Control Mode Switch Block

The Control Mode Switch Block is a special block present in the schedule. The Control Mode Switch Block is responsible for switching the Control Mode of a subsystem. For the change in the Control Mode to be reflected on the subsystem, two steps needs to be performed: retrieval of the parameter sets from the non-cachable memory to the cachable memory, and switching to the newly obtained parameter set. Certain number of samples are reserved in advance to perform the Control Mode Switch. The Control Mode Switch Block, in the first step, communicates with a subset of the WorkerBlocks, and requests them to retrieve the new parameter set. The communication between the Control Mode Switch Block and the WorkerBlocks is synchronous (blocking call), i.e., the Control Mode Switch Block is blocked until the control is returned from the WorkerBlocks. Once all the WorkerBlocks retrieve their parameter sets from the non-cachable to the cachable

memory, the Control Mode Switch Block in the second step, communicates with all the WorkerBlocks, and requests them to switch from an active parameter set to the parameter set that is to be made active. The result of this step is the change in the Control Mode. The communication between the Control Mode Switch Block and all the WorkerBlocks in this step is blocking as well.

2.6 Current Software Architecture

The software architecture consists of an Application layer, Host and multiple Workers as shown in the Figure 2.14. Application is an entity that lets the multiple ServoGroups operate together to achieve a higher level goal. For example, an application implements a scan action by controlling the lens, reticle and wafer stage ServoGroups to operate together. Application consists of per ServoGroup, the ServoGroupConfiguration file and Network Definition File. Host is reponsible for control mode switching and imposing the state-change behaviour on the WorkerBlocks present on the Worker. The Host consists of a Process Control Manager and multiple ServoGroups. Each of the ServoGroups can be deployed on a single Worker, and hence, cannot be shared among the Workers. Host communicates with the Workers, both during initialization and run-time. Every Worker consist of a Block Factory module, a Sequencer module and all the blocks present in a ServoGroup.

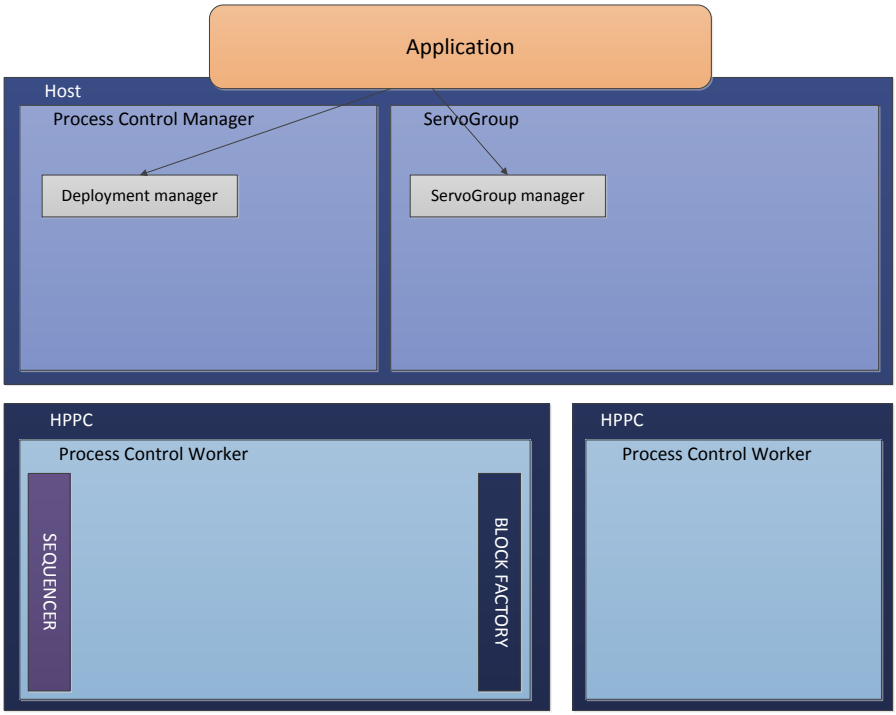

Figure 2.14: Software architecture

The Host consists of a process control manager and multiple ServoGroups. The Deployment Manager present in the Process Contol Manager is responsible for de-

ploying the ServoGroups on the Worker. The Deployment Manager reads the Network Definition file of every ServoGroup, and communicates with the Block Factory module, the WorkerBlocks to be created on the Worker and the connection between the WorkerBlocks, as shown in the Figure 2.15. The Block Factory module present on the Worker, creates the WorkerBlocks and establishes the connection between the WorkerBlocks, according to the information received from the Deployment Manager.

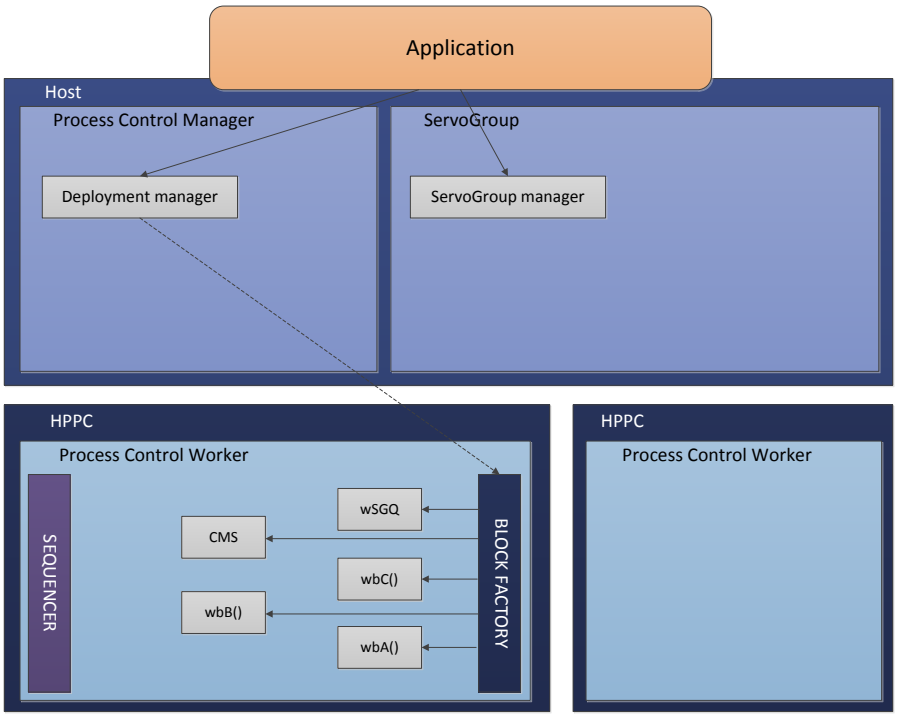

Figure 2.15: Deployment manager communicating with the Block Factory module

The Deployment Manager, also communicates with the Sequencer module present on the Workers. The Sequencer module, after receiving the schedule information, enforces the execution order on the WorkerBlocks as shown in the Figure 2.16. The communication between the Deployment Manager, Block Factory and Sequencer modules happens through Remote Procedure Call (RPC).

Once all the WorkerBlocks are created on the Workers, the ServoGroup Manager present in the ServoGroup, receives the addresses of the blocks from the Deployment manager, and creates the proxies of the blocks present on the Worker. The proxies of the blocks are created to communicate with their counterparts present on the Worker. Apart from the creation of the proxies, the ServoGroup Manager also creates the Control Modes according to the information received from the ServoGroup configuration file. Figure 2.17 shows the creation of the ServoGroup contents by the ServoGroup manager.

Figures 2.18 and 2.19 shows the message sequence diagrams of the communication between the Host, and Block Factory and Sequencer modules, and, the creation of the proxies by the Servo Group manager respectively.

Apart from the creation of different modules in the ServoGroup, the ServoGroup

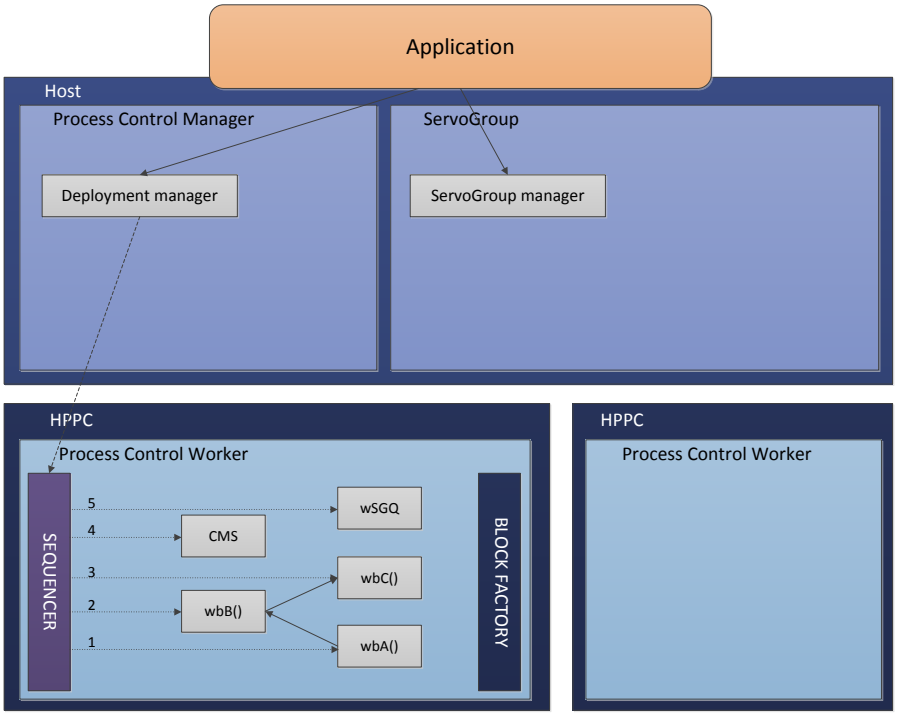

Figure 2.16: Deployment manager communicating with the Sequencer module

Manager also creates the parameter sets for each WorkerBlock. The ServoGroup Manager reads the ServoGroup Configuration file present in the Application, and creates the parameter sets per WorkerBlock on the proxies on the Host. The WorkerBlock proxies then load the parameter sets on their counterpart on the Workers, as shown in the Figure 2.17. Proxies on the Host communicate with the WorkerBlocks only during the initialization via Remote Procedure Call (RPC). Once all the parameter sets are communicated with the WorkerBlocks, the initial value of the parameter sets is loaded. Figure 2.20 shows the message sequence diagram of loading the parameter sets on the WorkerBlocks from the Host.

After the initial values of all the parameter sets are loaded on the WorkerBlocks, the Host signals all the WorkerBlocks to perform a Control Mode Switch. The Control Mode Switch is performed, by letting all the WorkerBlocks switch to the new parameter set. As soon as the Host receives the change in the Control Mode signal from the layer above it, the ServoGroup manager present on the Host maps the parameter set to the WorkerBlock, i.e., the ServoGroup manager identifies the parameter set that the WorkerBlock must choose to switch to the new Control Mode. Once the mapping of the parameter set to the WorkerBlock is done, the ServoGroup manager collects the mapping data, and communicates it with the Proxy ServoGroupQueue in the form of a packet. The Proxy ServoGroupQueue communicates the packet received from the ServoGroup manager with the Worker ServoGroupQueue asynchronously, via Remote Procedure Call (RPC), as shown in the Figure 2.21.

The Worker ServoGroupQueue decrypts the packet received and communicates this

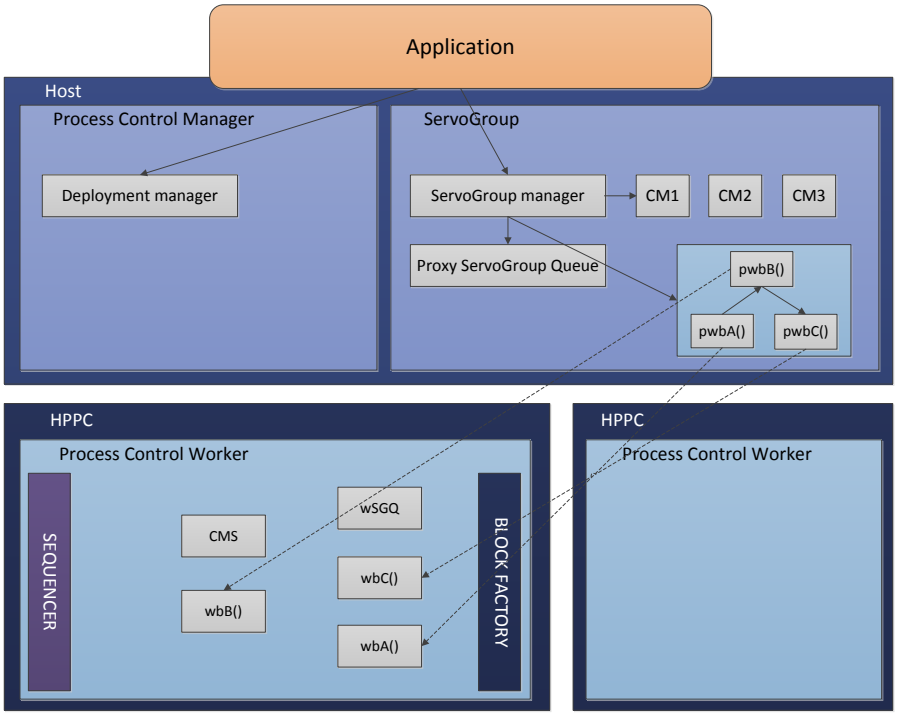

Figure 2.17: ServoGroup manager creates the contents of the ServoGroup

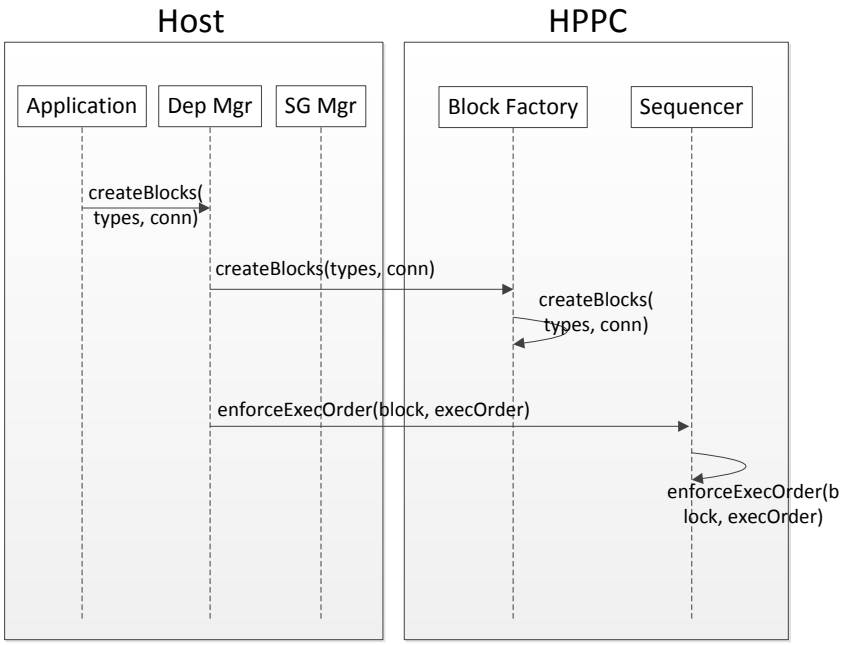

Figure 2.18: Initialization

information with the Control Mode Switch block. The Control Mode Switch block, after receiving the data from the Worker ServoGroupQueue block, communicates with

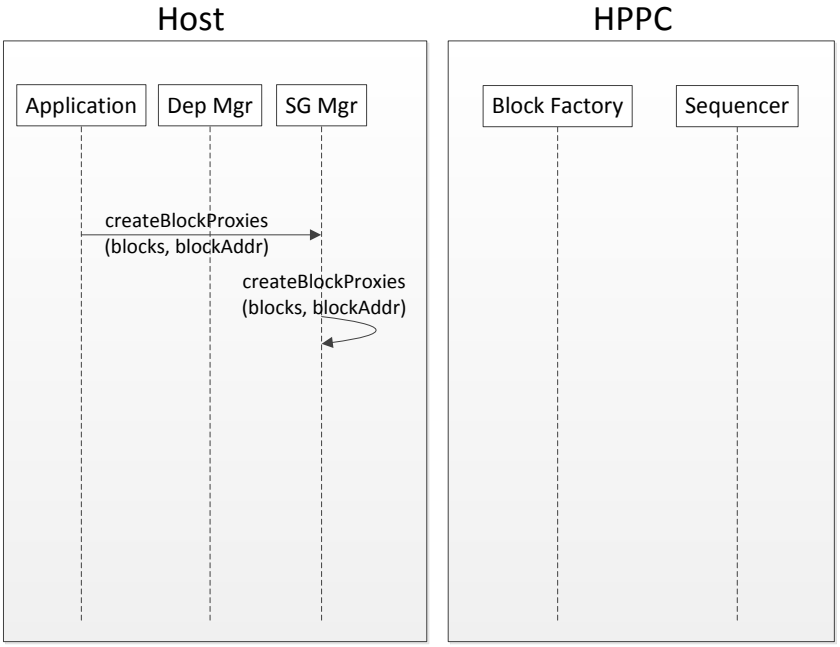

Figure 2.19: ServoGroup manager creates the proxies on the Host

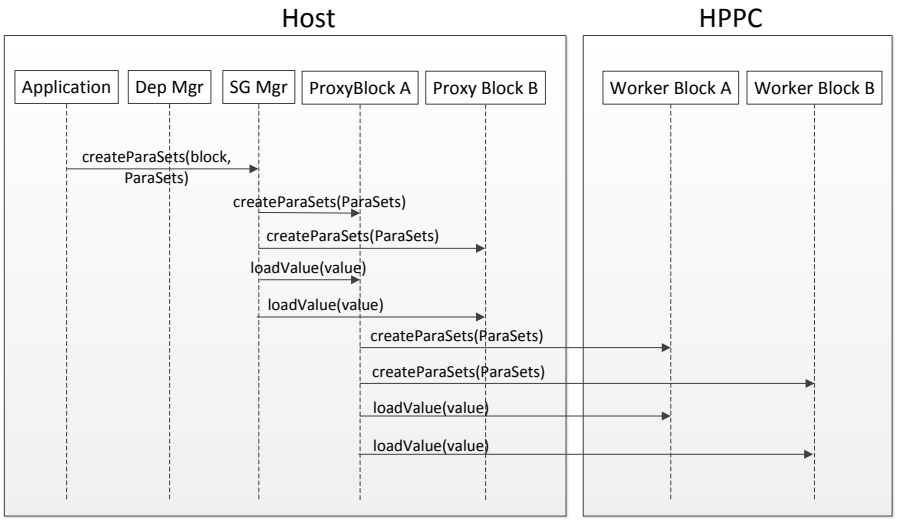

Figure 2.20: Creation of Parameter Sets

the WorkerBlocks, and signals to load the parameter set that is to be made active, from non-cachable memory to cachable memory. Figure 2.22 shows the communication of Control Mode Switch block with the WorkerBlocks. Parameter Sets loading from non-cachable memory to cachable memory happens during the background part of the sample. After certain number of samples, all the WorkerBlocks load the parameter sets from the non-cachable memory to the cachable memory. Number of samples fixed to load the parameter sets from non-cachable memory to cachable memory is 49. During the

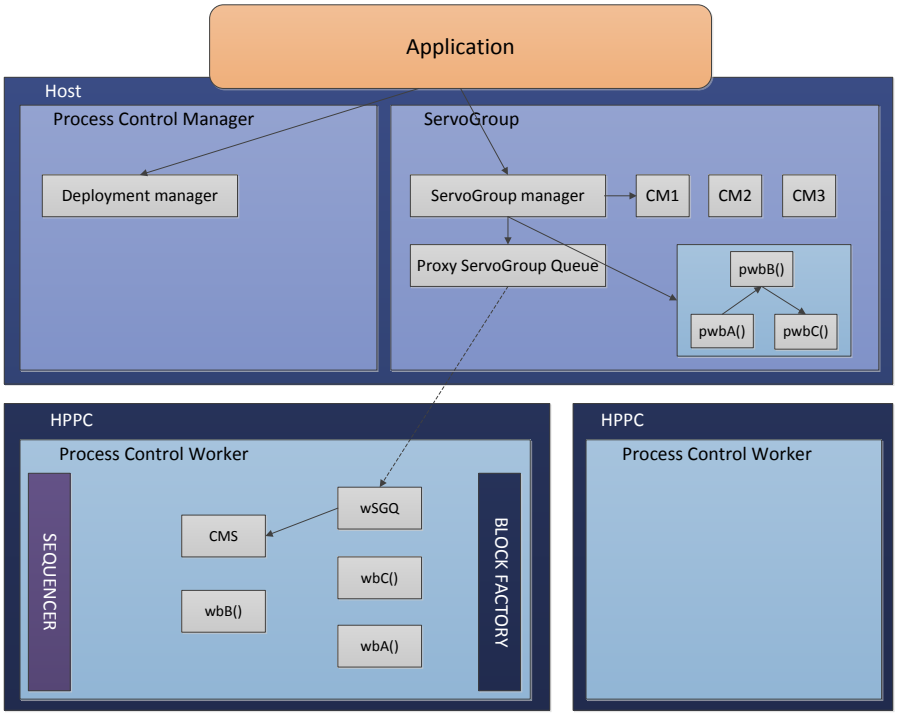

Figure 2.21: Asynchronous communication between Proxy ServoGroupQueue and Worker ServoGroupQueue

50th sample, the Control Mode Switch block, calls all the WorkerBlocks synchronously, to switch from the active parameter set to the parameter set that is to be made active, as a result of which, the subsystem switches to the new Control Mode.

Figure 2.23 shows the message sequence chart of the Control Mode Switching behaviour.

2.7 Current Hardware Architecture

The hardware architecture currently used in ASML consists of an ATCA rack, as shown in Figure 2.24 [3]. An ATCA rack consists of ATCA blades, which can be either processor blades, or communication blades. An ATCA rack generally consists of a single communication blade and multiple processor blades. All the blades in an ATCA rack are connected to one another via serial Rapid IO connectivity.

Every ATCA blade consists of a number of slots into which Advanced Mezzanine Cards (AMCs) can be placed, which contains one or more processors. The ATCA rack, as shown in Figure 2.24 consists of the following AMCs: Host, HPPCs and QHA. The Host/HPPCs are PowerPC based MPC8548E single-core processor and the QHA module is responsible for the communication between the AMCs and Sensors/Actuators. The Host connects to all the HPPCs via ethernet, however, all the HPPCs are connected to one another via serial Rapid IO connection. The serial Rapid IO connectivity consists of 4 lanes operating at a frequency of 2.5 GBaud/sec [8].

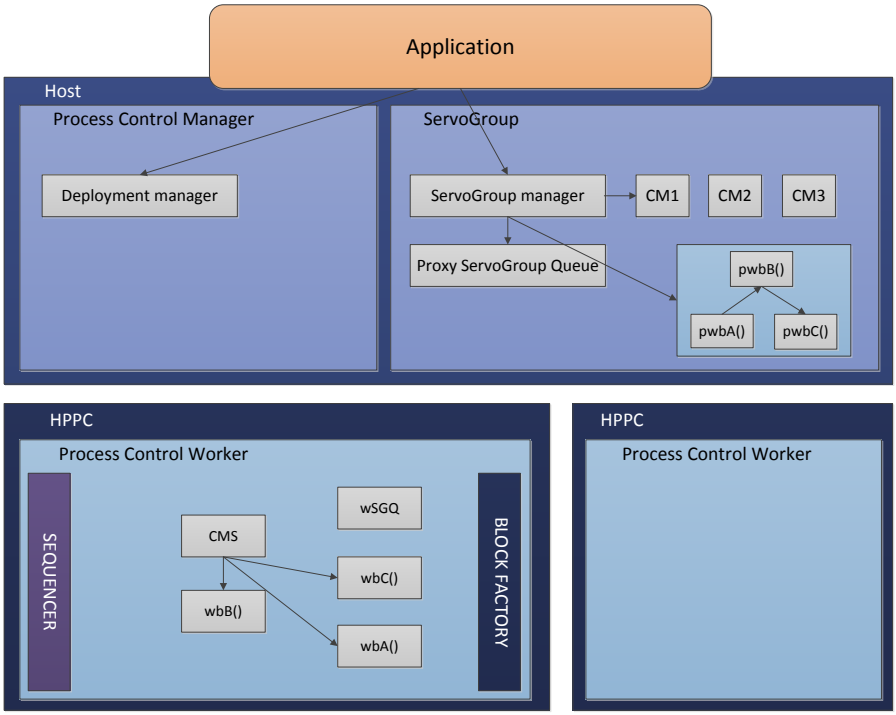

Figure 2.22: Control Mode Switch block signalling the WorkerBlocks to perform the Control Mode Switch

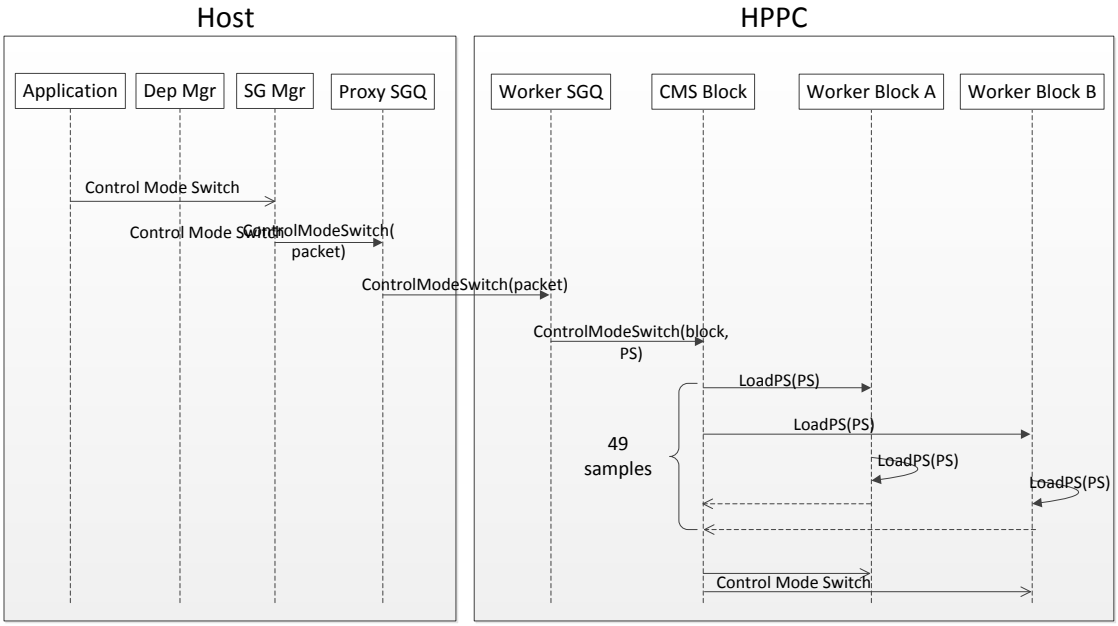

Figure 2.23: Control Mode Switch Behaviour

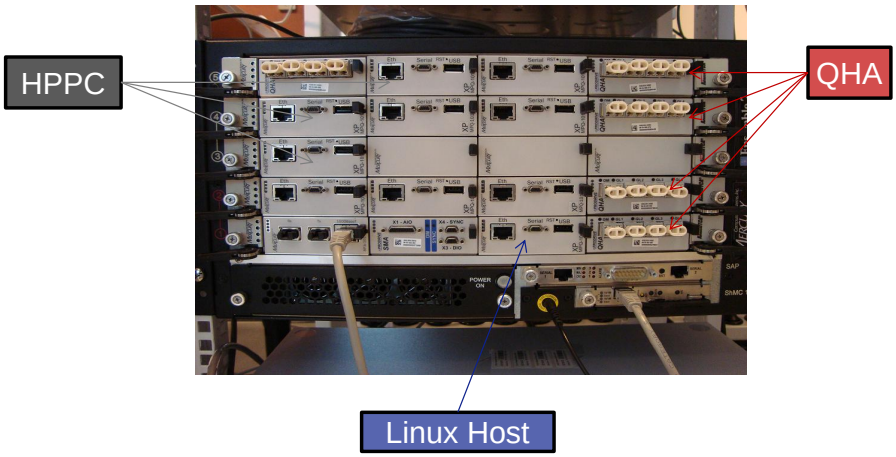

Figure 2.24: ATCA rack [3]

3 Previous Work

This chapter describes in detail the experiments carried out by Frijns et al [5] using FPGA as a platform for digital control applications.

The benchmark application discussed earlier was chosen for deployment on an FPGA. The hardware platform consisted of a high end Altera Stratix V GX FPGA and the benchmark application chosen was deployed on a multi-ASIP platform in FPGA. The performance of the benchmark applications obtained after deploying them on an FPGA were compared with the performance obtained after mapping them on a high-end 1.2GHz Freescale P4080 octo-core GPP.

An *ASIP* is abbreviated as an application specific instruction set processor, whose instruction set is tuned towards a certain class of applications. An ASIP provides a good trade-off between the flexibility offerred by general-purpose processors and performance offerred by ASICs (application specific integrated circuits). Vector processing units (VPUs), Scalar processing units (SPUs) and Look-up units (LUs) are examples of ASIPs. In case of multi-ASIP FPGA implementation, the control blocks are mapped on the ASIPs, connected to one another via switch as shown in the Figure 3.1. Every ASIP

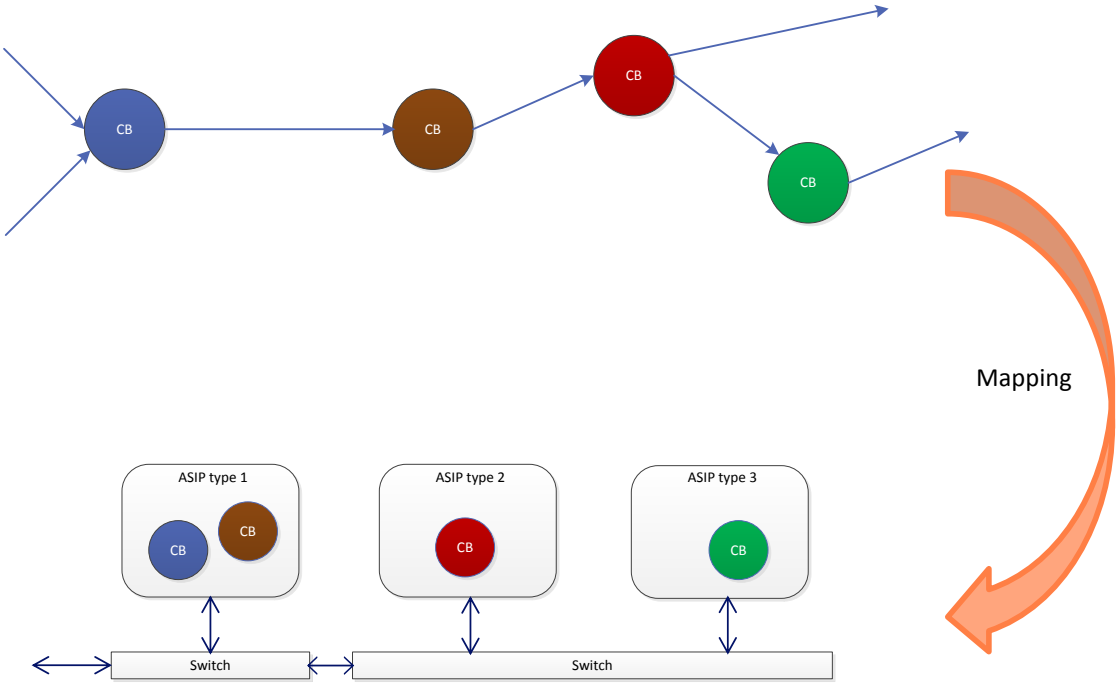

Figure 3.1: Control Blocks Deployed On ASIPs

has its own input memory, state memory and parameter memory as described in the

Figure 3.2. The presence of the separate memories decouples the communication from computation. ASIPs are mapped on an FPGA and are connected to each other using a

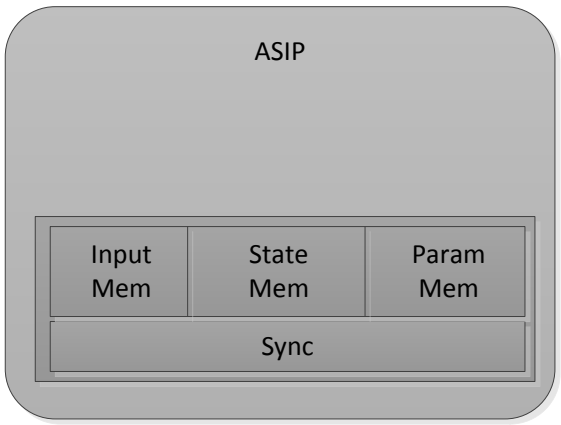

Figure 3.2: Application-specific instruction-set processor

switch as described in the Figure 3.3. The sampling frequency and IO delay obtained in case of multi-ASIP FPGA technique was 133 KHz and 10 μs respectively [5].

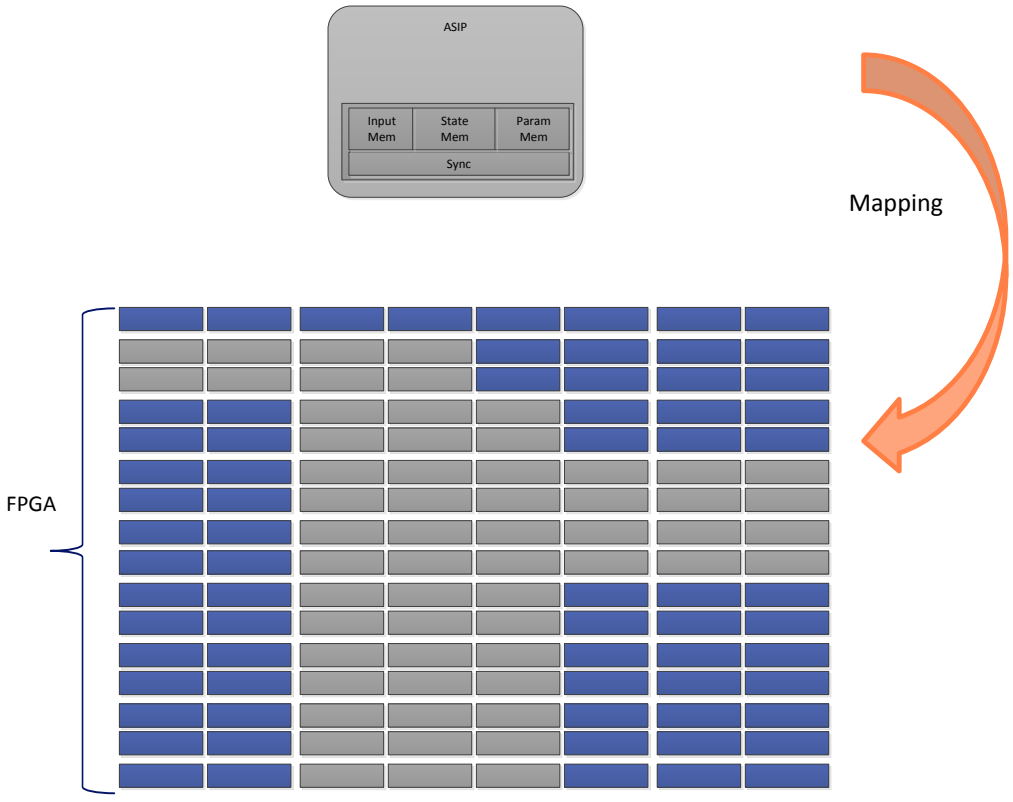

Figure 3.3: ASIP mapped on an FPGA

In case of spatial mapping approach, an application is deployed directly on an FPGA.

Figure 3.4 shows the mapping of the control blocks directly on an FPGA. The sampling frequency and IO delay obtained in case of spatial mapping approach was 154 KHz and 8.7 μs respectively [5]. Although the spatial mapping technique performed better than the multi-ASIP technique, the FPGA resources utilized by the spatial mapping technique was found out to be much more than the multi-ASIP technique. This means that the spatial mapping technique performs better, but at the cost of resources, whereas the multi-ASIP FPGA offers a better trade-off between the performance and resource utilization.

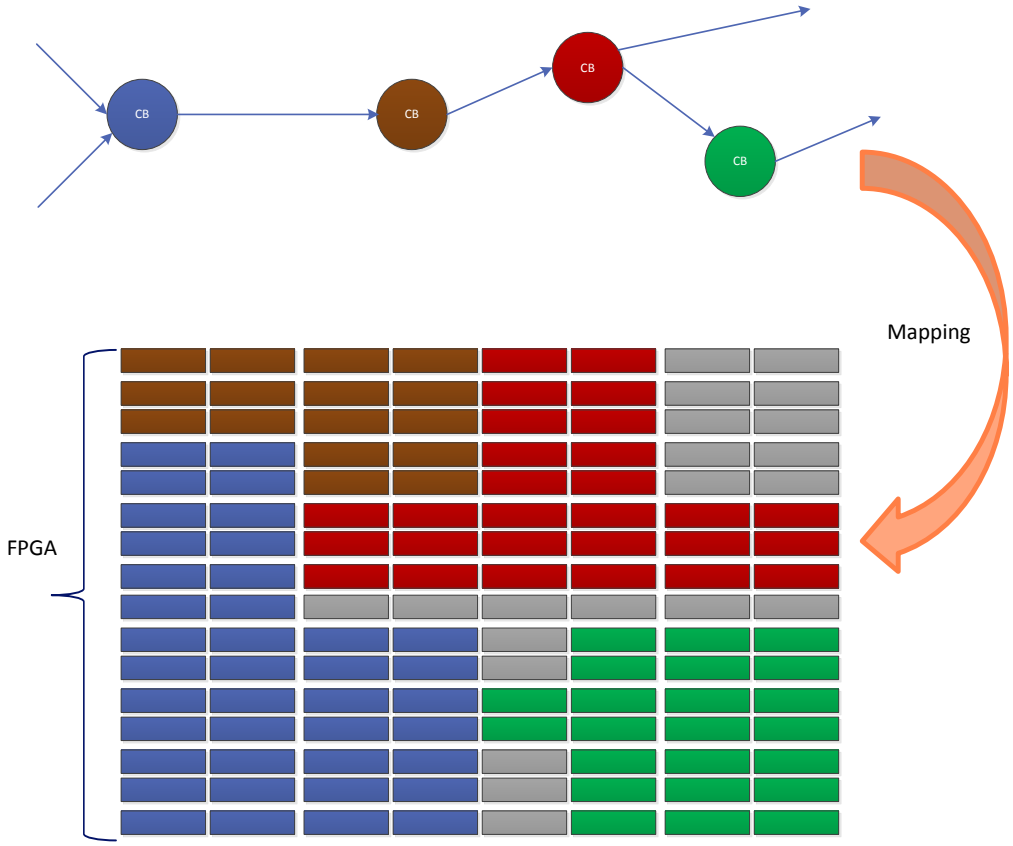

Figure 3.4: Control blocks mapped on an FPGA

4 Analysis

This chapter discusses software analysis and hardware analysis in detail. Based on the results obtained from software analysis, the decision of the part of the benchmark application that is to be offloaded on the FPGA is taken. Based on the results obtained from hardware analysis, the decision of the hardware development kit to be considered for the project is taken.

4.1 Software Analysis

As mentioned earlier, since the Short Stroke controller provides nano-meter accuracy and fine tunes the position of the wafer, it processes more data compared to the Long Stroke controller, and hence consists of control blocks of larger dimensions. One such block present in the Short Stroke controller is the state-space block, which is shown in the Figure 4.1 [4].

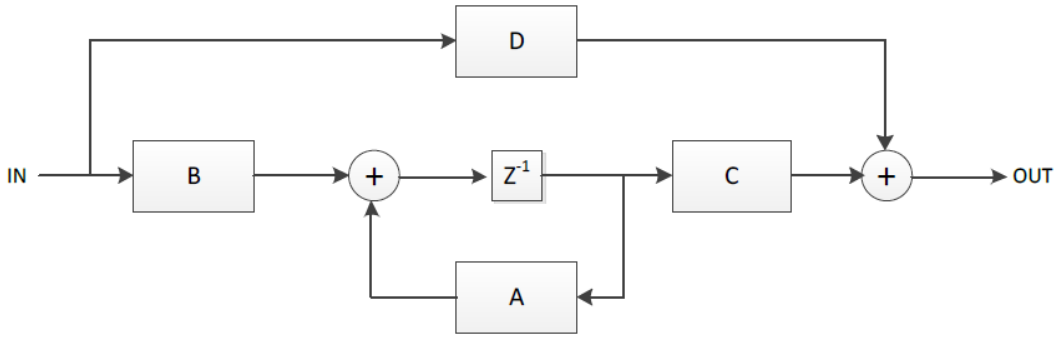

Figure 4.1: Block diagram representation of State-space model [4]

The matrix dimensions of a state-space block with X internal states, I inputs and O outputs is: $A(X, X)$, $B(X, I)$, $C(O, X)$, $D(O, I)$. It has been envisioned that the future Short Stroke controller will consist of state-space blocks with the 220 states, 12 inputs and 12 outputs. Hence, the matrix dimensions of the state-space block in the future Short Stroke controller will be: $A(220, 220)$, $B(220, 12)$, $C(12, 220)$, $D(12, 12)$.

As mentioned earlier, every control block has a fullCalc() function, which can be divided into preCalc() and postCalc() to reduce the IO delay between the sensors and the actuators. The fullCalc() of the state-space block is described as below:

$$\begin{bmatrix} X_{k+1} \\ O_k \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} X_k \\ I_k \end{bmatrix} \tag{4.1}$$

The preCalc() function of the state-space block can be written as:

$$X_{k+1} = AX_k + BI_k \tag{4.2}$$

$$pre_k = CX_{k+1} \tag{4.3}$$

The postCalc() function of the state-space block can be written as:

$$O_k = pre_{k-1} + DI_k \tag{4.4}$$

In the previous work carried out at ASML, it was found out that the state-space block present in the benchmark application consumed 91.87% of the execution time [1]. The execution time of the state-space block was measured both on a General Purpose Processor (GPP) and a Vector ASIP on an FPGA separately. Table 4.1 [1] shows the execution time of the preCalc() and the postCalc() functions of the state-space block on a GPP and Vector ASIP on an FPGA.

	Pre Execution Time	Post Execution Time
GPP	51.7 μs	0.403 μs
Vector ASIP on FPGA	7 μs	0.8 μs

Table 4.1: state-space block execution time [1]

It can be observed from the table that the preCalc() function takes more time on a GPP when compared to the FPGA, whereas, the postCalc() takes more time on the FPGA when compared to that on the GPP. The performance of the benchmark application can be increased by either offloading both the preCalc() and the postCalc() functions, or by only offloading the preCalc() function. Offloading preCalc() function can definitely improve performance, however, offloading the postCalc() will certainly not improve the performance, as it takes longer time to execute on an FPGA. Both the options are taken into consideration and a detailed analysis is carried out to estimate the time left to communicate the inputs and the outputs of the preCalc() and the postCalc() functions respectively.

The time left for the communication of the input and the output data in the non-time critical section, if preCalc() function is offloaded on an FPGA can be calculated as:

$$time_{freePre} = time_{preGPP} - time_{preFPGA} = 51.8 - 7 = 44.7\mu s \tag{4.5}$$

The calculation shows that the time left in the non-time critical section is 44.7 μs.

Offloading the preCalc() on the FPGA requires the communication of both the inputs and the outputs, whereas, offloading the postCalc() on an FPGA does not require the communication of the input, as, the postCalc() function is dependent on a preCalc() function for its input. However, offloading the postCalc() function requires the communication of output data back to the GPP. It can be seen from the above obtained timings that the communication of the output data of the postCalc() in the time critical

part of the sample would prove to be costly, as the IO delay is directly affected, whereas, communicating the input and the output data in the non-time critical part of the sample can improve the performance.

The amount of data that can be sent in the non-time critical part of the sample can be estimated by knowing the link delay between the two AMC cards and the link bandwidth. It has been estimated that the link latency between the two AMC cards is 1 μs and the link bandwidth is 250MB/s, which is equal to 250B/μs.The amount of data that can be transferred conisdering the above calculations is:

$$(time_{freePre} - 2 * 1\mu s) * 250B/\mu s = (44.7 - 2 * 1) * 250 \tag{4.6}$$

The value obtained from the above equation must be greater than the amount of input and output data communicated between the GPP and FPGA. So, the above equation leads to the figure of 10675 bytes, which is equivalent to 2668 32-bit values.

If the preCalc() function is offloaded on the FPGA, the input and the output data has to be communicated every sample between the GPP and FPGA. The maximum size of the input data in the benchmark application is 11 and the biggest output size is 12. It can be observed from the equation of the postCalc() function that it requires only one vector which is of the size of the output from preCalc(). Hence, the time saved per sample in the non-time critical section is:

$$((2668 - inputdata - outputdata) * 4)/250bytes/\mu s = ((2668 - 11 - 12) * 4)/250bytes/\mu s = 42.32\mu s \tag{4.7}$$

However, if both preCalc() and postCalc() functions are offloaded on an FPGA, the time saved per sample in the non-time critical sections is:

$$((2668 - inputdata) * 4)/250bytes/\mu s = ((2668 - 11) * 4)/250bytes/\mu s = 42.51\mu s \tag{4.8}$$

Offloading both preCalc() and postCalc() functions on an FPGA requires communicating the output of the postCalc() function in the time critical part of the sample. The time taken to communicate the output data per sample in the time critical part is:

$$(time_{freePost} - 1\mu s - outputdata/250bytes/\mu s = -0.4 - 1\mu s - (11 * 4)/250bytes/\mu s = -1.576\mu s \tag{4.9}$$

The time lost in the time critical part of the sample is: 1.57 μs. This means that there is a delay in communicating the data in the time critical part of the sample, which affects the IO delay.

Hence, from the above obtained results, it can be concluded that offloading only the preCalc() function leads to the saving of 42.32μs in the non-time critical part of the sample. However, offloading both preCalc() and the postCalc() functions leads to the saving of 42.51μs per sample in the non-time critical part, whereas, in the time critical part, a penalty of 1.57μs has to be paid, which affects the IO delay.

4.2 Hardware platform analysis

A hardware development kit is chosen based on the following factors:

The resources offered by the development kit should be more than that required by the benchmark application. The estimation of the amount of resources required by the benchmark application is shown in the section below.

The processing units are placed on a carrier blade, and all the processing units are connected to one another via serial Rapid IO connectivity, at a speed of 4*2.5 GBaud. The development kit under consideration should be at par with this to maintain the IO delay between the 2 processing units.

The development kit to be considered should be feasible with the existing CARM architecture. Currently, all the processing units are placed on the AMC cards on a carrier blade, where each of the AMC cards are connected via Rapid IO. Hence, the kit should be chosen such that should be possible to place it on an AMC cards on a carrier blade.

Finally, the development kit to be considered should be available in the market.

4.2.1 Hardware Platforms under consideration

Following are the hardware platforms that can be considered:

1. Graphics Processing Units (GPUs).
2. System on Chip (SoC), with a GPP and FPGA on it.
3. A separate FPGA.

A Graphics Processing Unit consists of many processors and hence can execute the data in parallel [9]. GPU can be considered as a development kit if the benchmark application consists of embarassingly parallel parts. Our benchmark application certainly consists of the blocks such as the state-space blocks which are embarassingly parallel. However, the latency to get data to GPU is huge, and moreover, GPUs consume more power. Also, choosing GPU as a development kit is not feasible with the existing CARM architecture, as it cannot be placed on an AMC card. All the above mentioned factors makes GPU a bad choice as a development platform.

System On Chip (SoC), with a General purpose processor and an FPGA on it can definitely be considered as an option [10]. Since, both the FPGA and the GPP are present on the same chip, the memory between the GPP and the FPGA is shared [11], and can be used for many purposes. The resources offerred by an SoC is estimated to be sufficient. However, it was estimated that the SoCs that can be used are out of stock, and that is one of the reasons, SoC is not chosen for the project. Using SoC as a development kit is left for the future work.

A separate FPGA as a development kit is another option that can be considered. Choosing FPGA as a development kit satisfies all the conditions listed above. An FPGA can be confiured to exploit the right mix of task-level and data-level parallelism. To this end it can be loaded with a mix of soft cores such as Vector ASIPs, Scalar ASIPs, Look-up ASIPs that can exploit task-level parallelism. The Vector ASIPs in turn can exploit data-level parallelism present in the control blocks. Typical mid-end to high-end

FPGAs have sufficient resources to map the required number of ASIPs to execute the application within its time constraints. The analysis of the resource estimates is shown in the section below. An FPGA is feasible with the CARM architecture, as it can be placed on an AMC card. These considerations make FPGA the platform of choice for this project.

4.2.2 FPGA - SoC Resource Estimates

The number of resources required depends on the number of processing units used. Below are the processing units that are required for our project:

1. 4 vector ASIPs for the calculation of 4 state-space blocks.
2. 1 SRIO unit for communication between the GPP and FPGA.
3. 1 Switch unit for the communication between ASIP cores within the FPGA.
4. 1 ASIP that controls the loading of the parameter sets from DDR3 SDRAM to parameter memory.

Since a state-space block is chosen to be deployed on a multi-ASIP FPGA, the resources required depends on the number of parameter sets and the size of each of parameter set. A state-space block consists of 4 matrices as parameters: Size of each parameter set would be:

$$((220 \times 220) + (220 \times 12) + (12 \times 220) + (12 \times 12)) \times 32(single - precision floating - point) = 1,722,368 bits$$

Size of two parameter sets would then be equal to: 3,444,736 bits.
The number of BRAM bits required by each of the processing units are estimated as below:

1. Vector ASIP:

$$4(VectorASIPs) \times 4,489,216(BRAMbits) = 17,956,864 bits \tag{4.10}$$

2. SRIO: 127,776 bits
3. Switch: 1,792 bits
4. ASIP to load the parameter sets: 192,000 bits

Hence, the total number of BRAM bits required are:

$$17,956,864 + 127,776 + 1,792 + 192,000 = 18,278,432 bits \tag{4.11}$$

Table 4.2 compares the number of BRAM bits required for the project with the resources offerred by the Altera development kit. The table shows that the Altera development kits will not satisfy the resource requirements of the project.

	Our Application	Altera Arria	Altera Cyclone	Altera Stratix	Altera Arria-SoC	Altera Cyclone-SoC
BRAM bits	18,278,432	4,477,824	8,211,456	9,383,040	Not available	6,191,000

Table 4.2: Comparing the benchmark application with Altera FPGAs and SoCs

	Our Application	Xilinx Spartan-6	Xilinx Artix-7	Xilinx Kintex-7	Xilinx Virtex-7	Xilinx Zynq-SoC
BRAM bits	18,278,432	4,800,000	13,000,000	34,000,000	68,000,000	27,180,000

Table 4.3: Comparing the benchmark application with Xilinx FPGAs and SoCs

In the table 4.3, the memory requirements of our project are compared with the memory resources offerred by the Xilinx development kits. The table shows that Xilinx Kintex-7, Xilinx Virtex-7 and Xilinx Zynq-SoC satisfy our demands. Since Xilinx Zynq-SoC is not stock in the near future, the choices left are Xilinx Kintex-7 and Xilinx Virtex-7 FPGAs. We make use of Xilinx Virtex-7 FPGA as a hardware development kit in our project.

4.3 Implementation approaches

There are two options to implement the communication functionality between the CPU and FPGA:

1. Communication of the parameter sets during initialization.
2. Communication of the parameter sets during run-time.

The analysis performed in the previous section show that the time taken to communicate one parameter set of a state-space block, which is 215,296 bytes takes 5259.87 μs. The time taken to communicate one parameter set of the four state-space blocks is 4*5259.87, which is equal to 21039.48 μs. Obviously, the time required to communicate the parameter sets during run-time, is much more. In this case, the communication time of the parameter sets (21039.48) exceeds the execution time of the state-space block on the CPU (308 micro seconds). Hence, this approach was not considered for implementation.

The analysis also shows that the round-trip time required to communicate the input data (48 bytes) and output data (48 bytes) is 1 μs, which is acceptable for implementation. Hence, the approach of communication of the parameter sets during initialization is chosen for implementation.

5 Implementation

This chapter describes the approach taken to implement the control mode switch functionality and the communication functionality between the CPU and FPGA. Every increment described in this chapter, is a small step that leads to the final implementation of the control mode switching functionality and the communication functionality between the CPU and FPGA. The test set-up used is shown in the Figure 5.1.

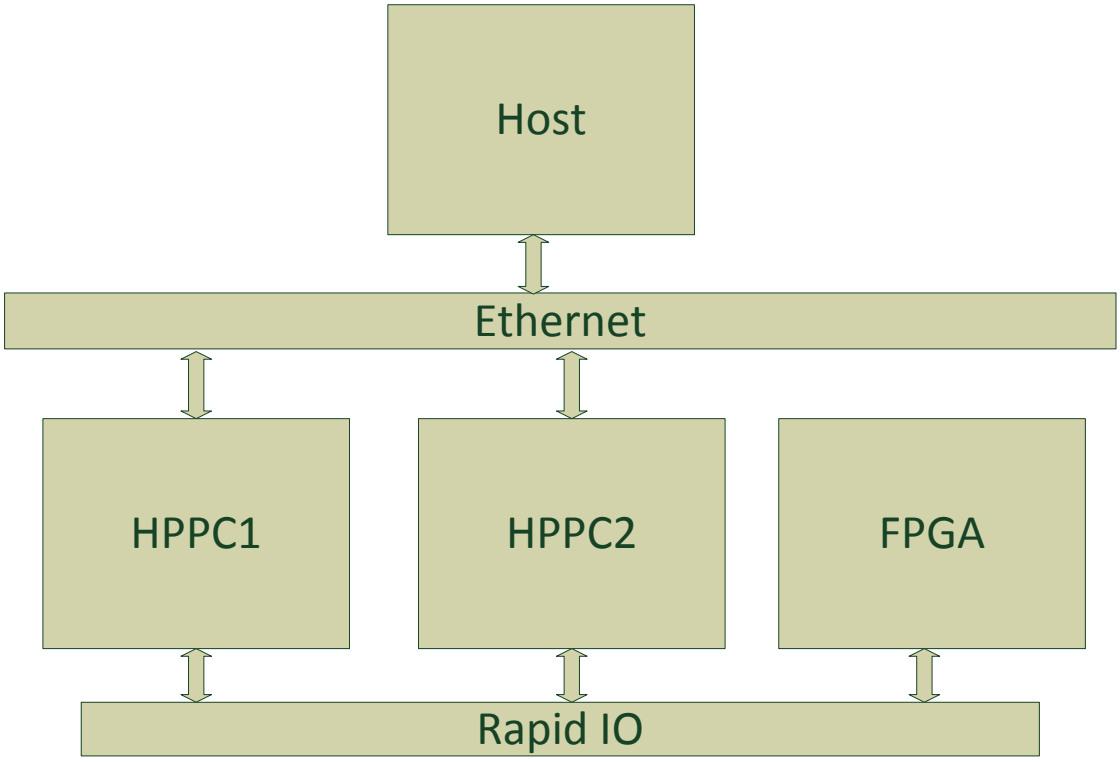

Figure 5.1: Test SetUp

5.1 Increment one

In this increment, all the blocks, present in the benchmark application are run on the Power PC, and the execution times of all the blocks are measured as a baseline for comparison.

5.2 Increment two

This increment involves programming the ASIP, communication of parameter set with the FPGA and also, the communication of data with the FPGA. No control mode switching is performed in this increment. In order to program an ASIP, the configuration data has to be written to the corresponding configuration address. The configuration data is generated by a code generator; which takes a specification of the platform, memory and application as inputs, and generates the configuration data. The configuration data obtained for the state-space block is read from the file generated by the code generator, and is written to the configuration address of the ASIP on the FPGA during initialization, as shown in the figure 5.2. The parameter sets communicated are written to the local memory of an ASIP. The message sequence diagram of communication of data is shown in the Figure 5.3.

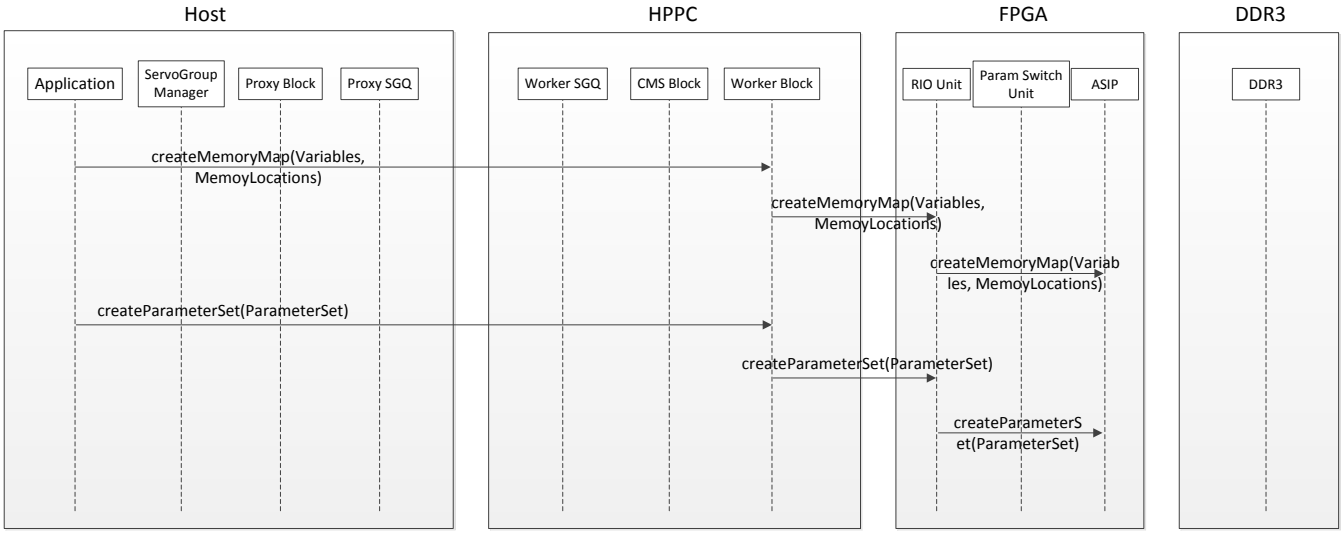

Figure 5.2: Increment Two - Initialization

5.3 Increment three

In this increment, a state-space block with 4 parameter sets is chosen for implementation. Hence, during initialization, one parameter set is written into the ASIP memory, and all the 4 parameter sets are written into the DDR3. Control mode switching is performed in this experiment. Since the parameter set of a state-space block is too large, it is difficult to accommodate all the parameter sets in ASIP memory. Hence, a single parameter sets is stored in ASIP memory and all 4 parameter sets in DDR3. Figure 5.4 shows the sequence diagram which describes the steps involved in the initialization. In this increment, 2 tasks are created for every block offloaded on the FPGA. Both the tasks include the same functionality, however, the parameter memory and the task addresses are unique. To simplify the control mode switching functionality, two tasks are created. When the control mode switch is signalled by the Host, one of the tasks still runs with the

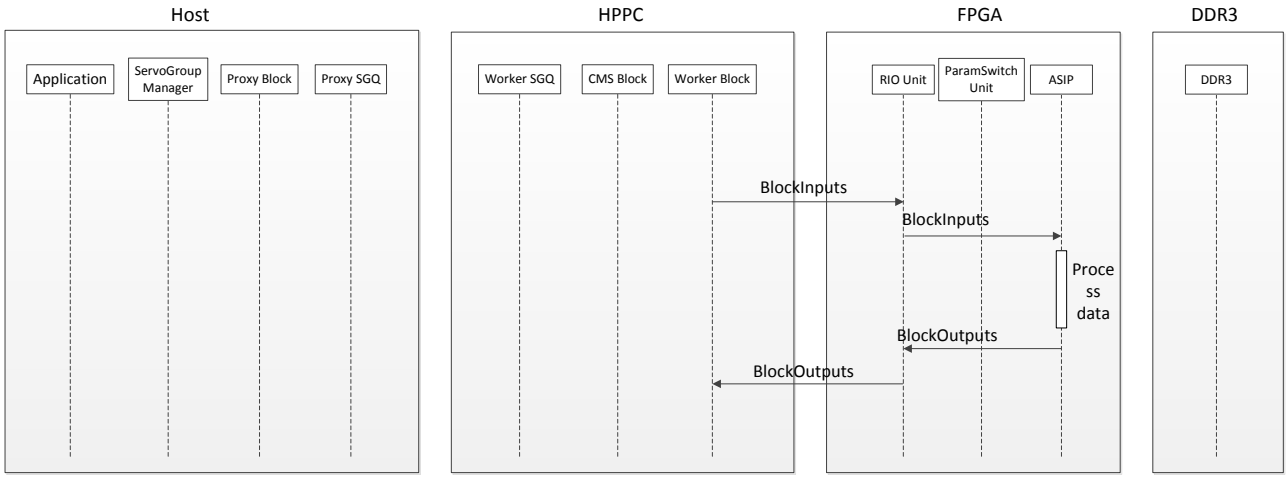

Figure 5.3: Increment Two - RunTime

old parameter set and inputs, while the parameter switch unit is loading a parameter set from DDR3 to the local memory on ASIP. When the parameter set is being loaded, no input data is communicated with that task. Once the task finishes loading the parameter set from DDR3 to local memory on ASIP, an acknowledgement is sent, and the input data is communicated with this task. Figure 5.5 shows the sequence diagram that describes the control mode switching functionality.

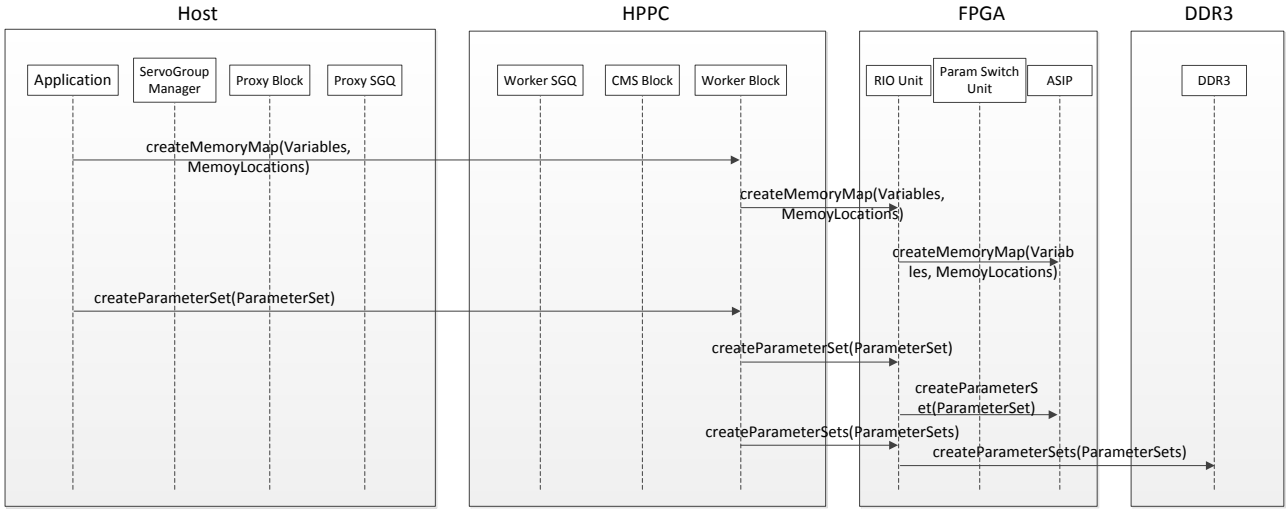

Figure 5.4: Increment Three - Initialization

At run-time, as explained in the increment two, input data is communicated with the FPGA, as shown in the Figure 5.5. Since a state-space block has 4 parameter sets, control mode switching functionality is performed at run-time. Once the WorkerBlock on the HPPC receives the Control Mode Switch command, the WorkerBlock communicates the parameter set Load command with the FPGA, as a result of which, the parameter

set is loaded from DDR3 into the memory in ASIP.

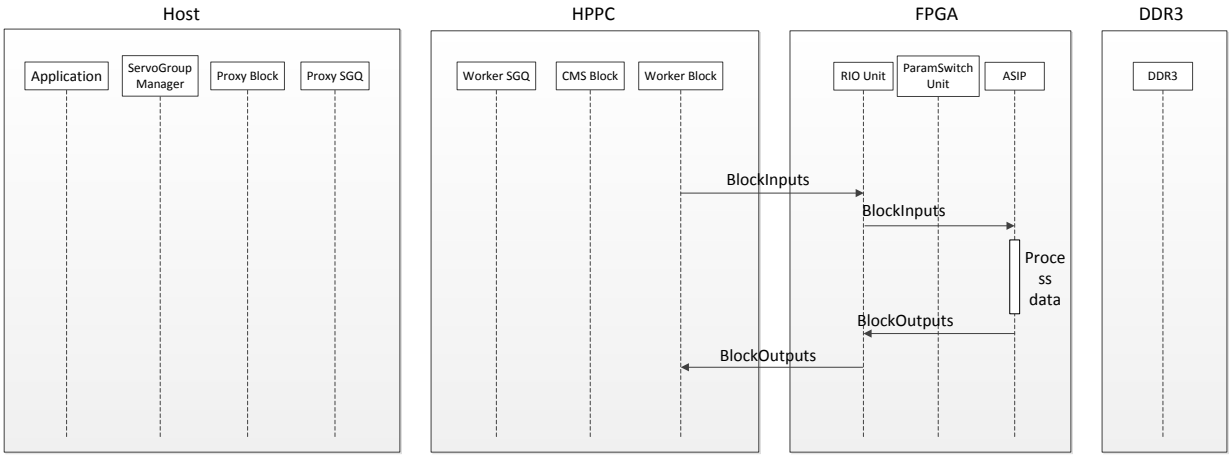

Figure 5.5: Increment Three - Run Time Scenario One

Once the parameter set is loaded from DDR3 into ASIP memory, an acknowledgement is sent to the WorkerBlock. After receiving the acknowledgement, the WorkerBlock starts sending the input data to the newly specified address on the FPGA, as shown in the figure 5.6.

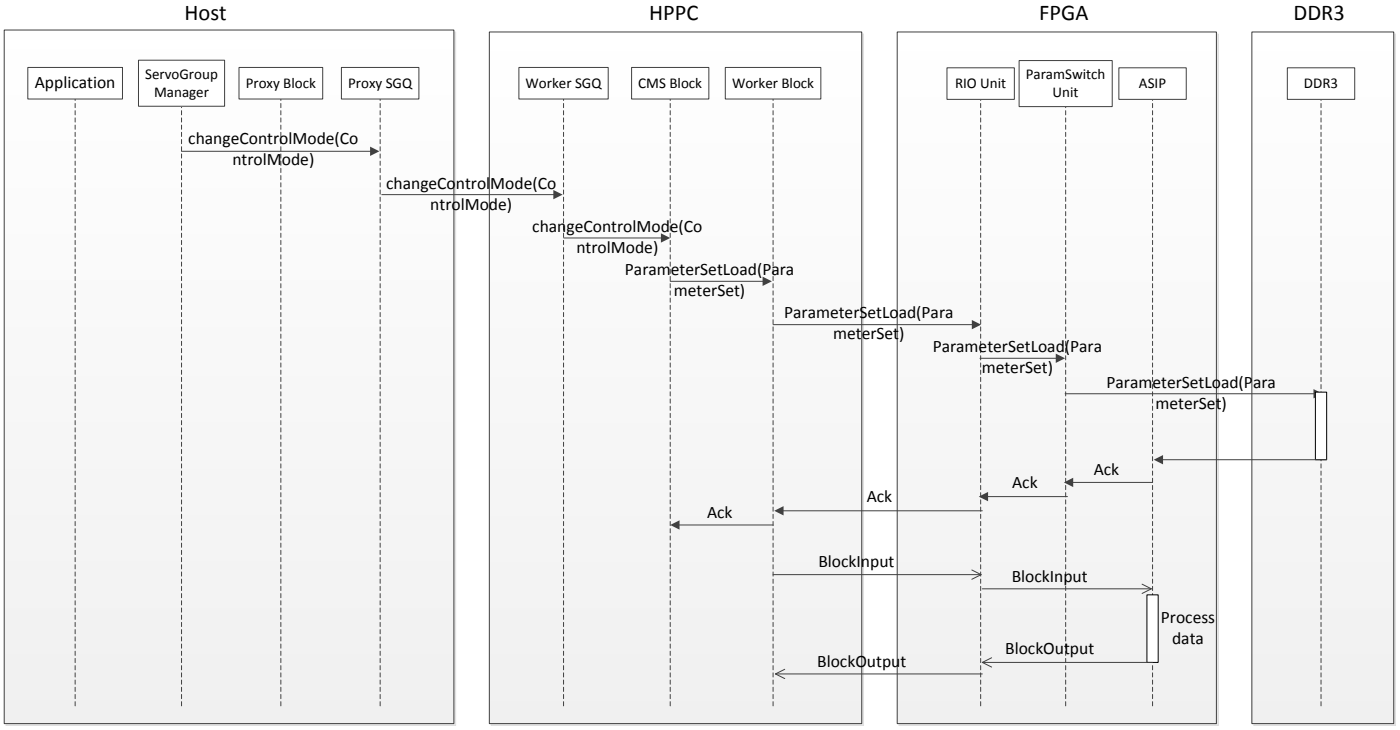

Figure 5.6: Increment Three - Run Time Scenario Two

6 Experiments

This chapter discusses the various experiments and the results in detail. All measurements are carried out on a MPC8548E PowerPC single-core processor clocked at 1.33 GHz. Section 6.1 discusses the measurements of a benchmark application on a PowerPC, Section 6.2 describes the measurements on a single core processor when used in combination with the VPEs. Similarly, Sections 6.3 to 6.7 shows the measurements on various cores on combination with the VPEs. In Section 6.8, experiments are performed by varying the number of inputs, outputs and states, and finally in Section 6.9, control mode switching measurements are described in detail.

6.1 Benchmark application measurements on PowerPC

The execution times of the blocks in the benchmark application are shown in Table 6.1. The execution times are measured using *clock_gettime()* function present in the *time.h* library. Each of the control block is executed 1 million times and the total execution time is recorded using *clock_gettime()*. The total time recorded is then divided by 1 million to obtain the average execution time of the block. This procedure is repeated 10 times and the final execution time is divided by 10 to obtain the execution time of the block. The data communication time is obtained by communicating the data with the second processing unit and not an FPGA. This is because the FPGA implementation was not completed when these experiments were performed. An FPGA was replaced by a processing unit and a functional simulator is created. The job of the simulator is to receive the data, verify the contents of the data and write back the output to the sending processing unit.

In the first experiment, all blocks are executed on a single core PowerPC without any FPGA acceleration, then the sum of all block execution times obtained is 620,022 ns. The achievable sampling frequency is the reciprocal of the total execution time, which is 1.61 kHz.

6.2 Single core PowerPC + VPEs

In this experiment, a number of vector processing units in FPGA are used to offload the state-space blocks from the PowerPC. The issue width of these processing units can be 32 or 64, and upto 4 of these can be used.

The execution time of a state-space block on a 32 wide VPE is 11.30 μs and on a 64 wide VPE is 8.08 μs. In order to measure the execution time and the sampling frequency of the benchmark application, we have to first calculate the execution time of the benchmark application without considering the preCalc functionality of the state-

Blocks	Execution time (ns)
CO_SPG_FF(6)	518.04
CO_PID_LP(6)	258
FLT_8(12)	938.88
MAT_6x2	27.21
AS_GS_6x6	143.51
AS_LOS_DC	36.94
MAT_8x6	144.96
MS_LOS2BM_COMBI	246.28
MS_LOS2BM_VER	89.47
MS_LOS2BF_COMBI	75.83
SS_220_preCalc(4)	612,714
SS_220_postCalc(4)	3011.44
CO_VAR_GAIN	9.92
SATURATION	318.44
AS_SS_EMDC	159.22
AS_SS_GS	428.46
MAT_13x11	455.64
MAT_12x12	446.70
Total time	620,022 ns

Table 6.1: Profiling results of the constituent blocks of the benchmark application measured on a 1.33 GHz single-core PowerPC

space block, which is described in the equation below.

$$
\begin{aligned}
ExecTime_{NoSSBlock} &= ExecTime - (4 * ExecTimeOfSSBlock) \\
ExecTime_{NoSSBlock} &= 620,022.94 - (4 * 153,178.5) \\
ExecTime_{NoSSBlock} &= 7308ns = 7.30\,\mu s = 0.007ms
\end{aligned} \tag{6.1}
$$

6.2.1 1 VPE32 and 1 VPE64

In this case a single 32 wide VPE and a single 64 wide VPE is used to offload all four state-space blocks on the VPE. These blocks are executed sequentially on the VPE. In order to calculate the execution time, we have to consider the communication time between the CPU and FPGA, and the time required for the data to reach from the RIO unit of the receiving data to the VPE and back from VPE to the RIO unit of the receiving end. The round-trip communication time between the CPU and FPGA, for the number of inputs and number of outputs of the state-space block present in the current benchmark application is 1 μs, and the additional time required for the data to travel from RIO unit to VPE and back from VPE to RIO unit is 0.164 μs. The execution time of the benchmark application obtained in this case is shown below. We do not show the calculations for VPE64 here, as it is similar to the calculations shown for VPE32. The

sampling frequency obtained in case of VPE64 can be seen in the Table 6.2.

$$ExecTime_{OneVPE32} = (ExecTime_{NoSSBlock}) + (4 * round - triptime) + (4 * TimeFromRIOUnitToVPE) + (4 * SSBlockExecTimeOnVPE32)$$
$$ExecTime_{OneVPE32} = 7.30 + (4 * 1) + (4 * 0.164) + (4 * 11.30)$$
$$ExecTime_{OneVPE32} = 56.74\,\mu s = 0.056ms \quad (6.2)$$

$$SampFreq_{OneVPE32} = 1/ExecTime_{OneVPE32} = 1/0.056 = 17.85kHz \quad (6.3)$$

6.2.2 2 VPE32 and 2 VPE64

This scenario considers two 32 wide and two 64 wide VPEs, and we deploy 4 state-space blocks on 2 VPEs. Since we have 2 VPEs, we deploy and execute 2 state-space blocks in parallel. Since there is only one communication unit between the CPU and FPGA, the input data have to be communicated with both the blocks on the VPE32 separately. The execution time and the sampling frequency of the benchmark application is shown below. Similarly, the sampling frequency of the benchmark application when the state-space blocks are deployed on 64 wide VPEs can be seen in the Table 6.2.

$$ExecTime_{TwoVPE32} = (ExecTime_{NoSSBlock}) + (4 * round - triptime) + (4 * TimeFromRIOUnitToVPE) + (2 * SSBlockExecTimeOnVPE32)$$
$$ExecTime_{TwoVPE32} = 7.30 + (4 * 1) + (4 * 0.164) + (2 * 11.30)$$
$$ExecTime_{TwoVPE32} = 34.53\,\mu s = 0.034ms \quad (6.4)$$

$$SampFreq_{TwoVPE32} = 1/ExecTime_{TwoVPE32} = 1/0.034 = 29.41kHz \quad (6.5)$$

6.2.3 4 VPE32 and 4 VPE64

Finally, we consider four 32 wide and four 64 wide VPEs, and we deploy all the 4 state-space blocks on the VPEs parallely. However, as explained above, the data has to be communicated with all the 4 state-space blocks on the VPE separately. Since, all the 4 state-space blocks can be scheduled on the VPE in parallel, the blocks on the CPU can continue with their execution without waiting for the reply from the state-space block on VPE. In the calculations below we consider the best case approximations when deploying the state-space blocks on the VPEs. The execution time of the benchmark application and the sampling frequency respectively is described below. The sampling frequency obtained when 64 wide VPEs used are shown in the Table 6.2.

$$ExecTime_{FourVPE32} = max(ExecTime_{NoSSBlock}, ((4 * round - triptime) + (4 * TimeFromRIOUnitToVPE) + (1 * SSBlockExecTimeOnVPE32)))$$
$$ExecTime_{FourVPE32} = max(7.30, ((4 * 1) + (4 * 0.164) + (1 * 11.30)))$$
$$ExecTime_{FourVPE32} = max(7.30, 15.956)$$
$$ExecTime_{FourVPE32} = 15.956\,\mu s = 0.015ms \quad (6.6)$$

$$SampFreq_{FourVPE32} = 1/ExecTime_{FourVPE32} = 1/0.015 = 66.67kHz \quad (6.7)$$

Table 6.2 shows the sampling frequency of the benchmark application when deployed on a single core processor and VPEs. It can be observed from the table that the hypothesis can be justified when the state-space blocks are deployed on 4 VPE32 and 4 VPE64.

	No VPE	**1 VPE32**	**2 VPE32**	**4 VPE32**	**1 VPE64**	**2 VPE64**	**4 VPE64**
Max achievable frequency (kHz)	1.61	17.85	29.41	66.67	22.72	35.71	83.33

Table 6.2: Sampling frequency of the benchmark application after deploying the state-space blocks on a single core processor and VPEs

Figure 6.1 shows the sampling frequency of the benchmark application after deploying the state-space blocks on VPE32 and VPE64 units.

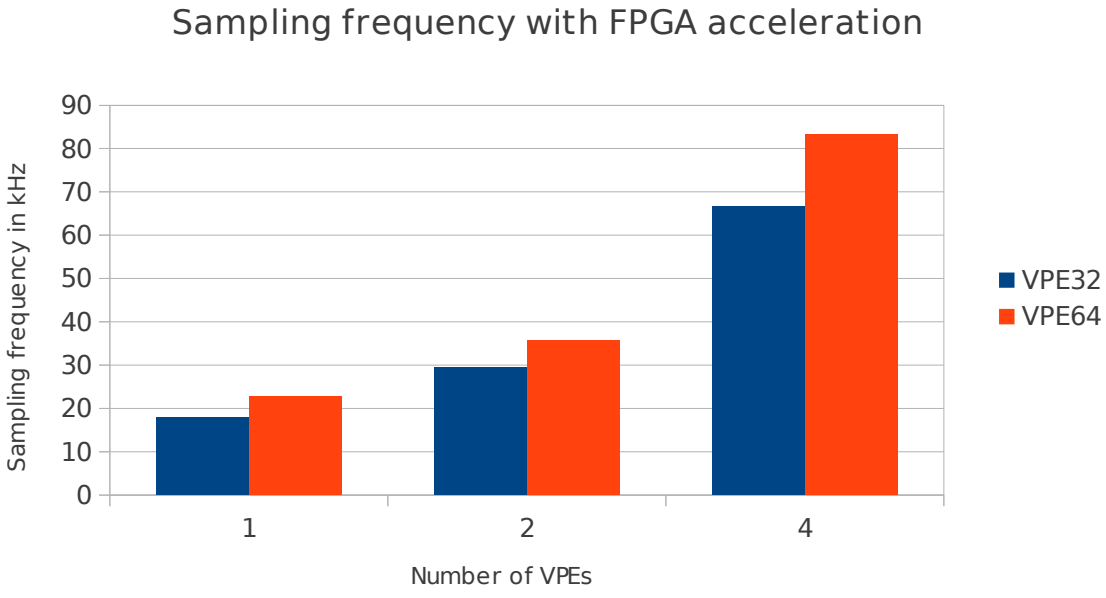

Figure 6.1: Sampling frequency of the benchmark application after deploying the state-space block on VPE32 and VPE64

6.3 Dual-core PowerPC + VPEs

In the previous sections, we performed the experiments on a single-core processor along with 32 wide and 64 wide VPEs. In this section, we perform the experiments on a dual-core processor along with 32 wide and 64 wide VPEs.

6.3.1 Dual-core

Here, we consider a PowerPC with 2 cores and perform the experiments. Since there are 2 cores, the benchmark application can be split among the cores for execution. On one of the cores, the Long Stroke controller is deployed, and since, the execution time of the Long Stroke controller is significantly smaller than the execution time of the Short Stroke controller, the Short Stroke controller can be deployed on both the cores, i.e., the core, on which the Long Stroke controller is deployed can be used to run the blocks present in the Short Stroke controller. The four state-space blocks in the Short Stroke controller can be split across the cores, i.e., two state-space blocks can be executed on one core and the other two state-space blocks can be executed in parallel on the second core. We do not consider the communication time between the cores, as it is negligible. The execution times on the first and second core can be calculated as shown below.

$$\begin{aligned} ExecTime_{FirstCore} &= ExecTime_{LongStroke} + (ExecTime_{ShortStroke})/2 \\ ExecTime_{FirstCore} &= 2.479 + 308,771.5 \\ ExecTime_{FirstCore} &= 308,773.97ns \end{aligned} \tag{6.8}$$

$$\begin{aligned} ExecTime_{SecondCore} &= ExecTime_{ShortStroke}/2 \\ ExecTime_{SecondCore} &= 308,771.5ns = 308.77\,\mu s \end{aligned} \tag{6.9}$$

The execution time and the sampling frequency of the benchmark application after executing it on 2 cores is as shown below. The sampling frequency obtained in this scenario is double the sampling frequency of that obtained in case of single core processor. This is because, the two cores can now be utilized to run the blocks parallely, i.e., we equally distribute the blocks on the cores.

$$\begin{aligned} ExecTime_{DualCore} &= max(ExecTime_{FirstCore}, ExecTime_{SecondCore}) \\ ExecTime_{DualCore} &= max(308,773.97, 308,771.5) \\ ExecTime_{DualCore} &= 308.773\,\mu s = 0.308ms \end{aligned} \tag{6.10}$$

$$SampFreq_{DualCore} = 1/ExecTime_{DualCore} = 1/0.308 = 3.24kHz \tag{6.11}$$

6.3.1.1 32 wide and 64 wide VPEs

In this scenario, the experiments are performed by deploying the state-space blocks on 32 wide and 64 wide VPEs. In the first experiment, the benchmark application is deployed on 2 cores; Long Stroke controller on first core and Short Stroke controller is split across the first and second core, and further, the 4 state-space blocks are deployed on a single 32 wide VPE or a single 64 wide VPE for execution. The Long Stroke controller and the Short Stroke controller are executed in parallel, and the 4 state-space blocks are executed sequentially on a VPE, as there is only one VPE considered for this experiment.

In the second experiment, two cores and two 32 wide VPEs or two 64 wide VPEs are considered. The 4 state-space blocks are deployed in parallel on 2 VPEs, and the other blocks are executed in parallel on the 2 cores.

In the final experiment, two cores and four VPEs are considered. We deploy the four state-space blocks in parallel on four VPEs, and the other blocks can be executed in parallel on the two cores. The sampling frequencies obtained after performing all the above mentioned experiments can be seen in the Table 6.3.

	No VPE	**1 VPE32**	**2 VPE32**	**4 VPE32**	**1 VPE64**	**2 VPE64**	**4 VPE64**
Max achievable frequency (kHz)	3.24	18.51	31.25	66.67	24.39	40	83.33

Table 6.3: Sampling frequency of the benchmark application after deploying the state-space blocks on a dual core processor and VPEs

It can be observed from the above table that the hypothesis can be justified by using 4 VPE32, 2 VPE64 and 4 VPE 64 respectively. Figure 6.2 shows the sampling frequency of the benchmark application when the state-space blocks are deployed on VPE32 and VPE64.

6.4 Tri-core PowerPC + VPEs

In this section, we carry out the experiments by considering 3 cores and VPEs 32 wide and 64 wide. The execution time and sampling frequencies are obtained by varying the number of VPEs from 1 to 4.

6.4.1 Tri-core

Here, in this case, we deploy the Long Stroke controller on one core and the Short Stroke controller is split across the three cores. The four state-space blocks present in the Short Stroke controller can now be executed on the three cores in parallel. Two state-space blocks can be executed on one core, and the other two state-space blocks can be executed in parallel on the other two cores. The execution time and the sampling

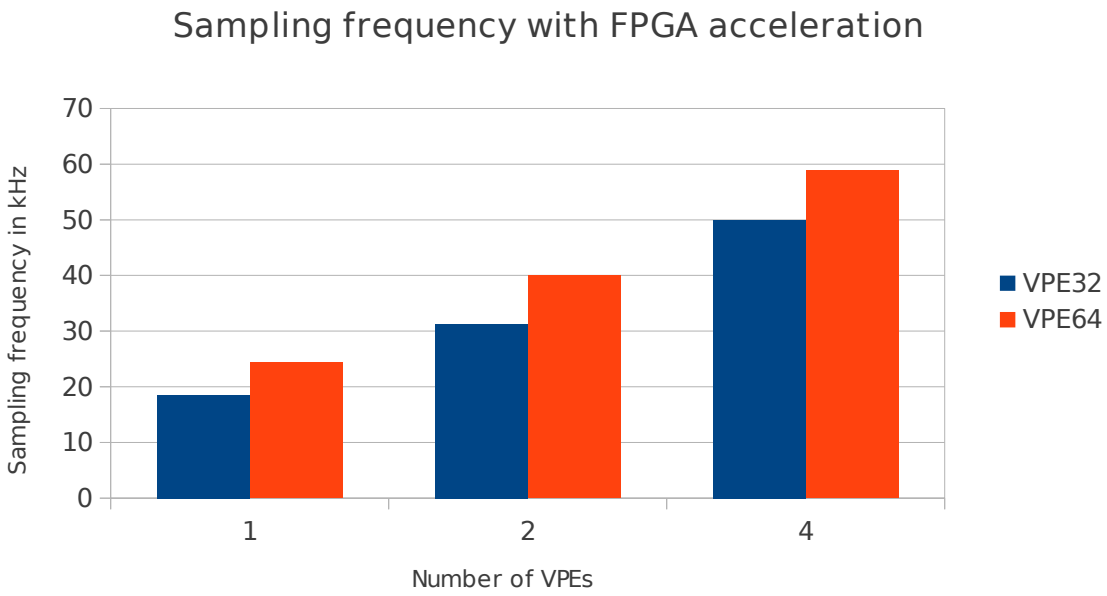

Figure 6.2: Sampling frequency of the benchmark application after deploying the benchmark application on two cores and state-space block on VPE32 and VPE64

frequency obtained in this case can be calculated as shown below. The execution time and sampling frequency of the benchmrk application after deploying it on a tri-core processor is as shown below. From the calculations, it can be observed that there is no much difference when compared to the sampling frequency obtained in case of dual core processors. The reason behind that is explained in the calculations below.

$$\begin{aligned} ExecTime_{TriCore} &= max(ExecTime_{(SSBlockOn1stCore} + ExecTime_{LongStroke}), \\ &\quad (ExecTime_{SSBlockOn2ndCore} + (ExecTime_{OtherBlocks})/2), \\ &\quad (ExecTime_{SSBlockOn3rdCore} + (ExecTime_{OtherBlocks})/2)) \\ &\quad + ExecTime_{SSBlock} \\ ExecTime_{TriCore} &= max(153180.97, 153180.64, 153180.64) + 153178.5 \\ ExecTime_{TriCore} &= 153180.97 + 153178.5 \\ ExecTime_{TriCore} &= 306.359\,\mu s = 0.306ms \end{aligned} \tag{6.12}$$

$$SampFreq_{TriCore} = 1/ExecTime_{TriCore} = 1/0.306 = 3.267kHz \tag{6.13}$$

6.4.1.1 32 wide and 64 wide VPEs

Here, we consider 3 cores along with 32 wide and 64 wide VPEs for experiments. In the first experiment, 3 cores and one 32 wide VPE and one 64 wide VPE is considered. The state-space blocks present in the Short Stroke controller can now be deployed on

the VPE in a sequential manner. Since all the other blocks present in the Short Stroke controller are executed in parallel, their execution times are not considered in the formula. The sampling frequency of the benchmark application when the state-space blocks are deployed on VPE32 and VPE64 can be seen in the Table 6.4.

In the second experiment, we consider three cores and two 32 wide and two 64 wide VPEs. Two of the four state-space blocks can be executed at a time on 2 VPEs. The sampling frequency obtained in this case is shown in the Table 6.4.

In this experiment, we consider three cores along with four 32 wide and four 64 wide VPEs. All the four state-space blocks can be executed on four VPEs in parallel. The sampling frequency obtained in this case is shown in the Table 6.4.

	No VPE	**1 VPE32**	**2 VPE32**	**4 VPE32**	**1 VPE64**	**2 VPE64**	**4 VPE64**
Max achievable frequency (kHz)	3.26	20.40	37.03	66.67	27.77	50	83.33

Table 6.4: Sampling frequency of the benchmark application after deploying the state-space blocks on a tri core processor and VPEs

It can be seen from the above experiments that, the combination of three cores + 4 VPE32, three cores + 2 VPE 64 and three cores + 4 VPE64 justifies the hypothesis claimed. Figure 6.3 shows the sampling frequency of the benchmark application obtained graphically.

6.5 Quad-core PowerPC + VPEs

In this section, we perform the experiments on a dual-core processor in combination with the 32 wide and 64 wide VPEs. We vary the number of VPEs from 1 to 4 in these experiments.

6.5.1 Quad-core

The benchmark application is deployed across the four cores in this experiment. The Long Stroke controller is deployed on one core and since, the execution time of the Long Stroke controller is significantly smaller than the Short Stroke controller, the Short Stroke controller is split across all the cores, including the one, on which the Long Stroke controller is deployed. The four state-space blocks present in the Short Stroke controller are executed in parallel on all the four cores. The execution time and the sampling frequency of the benchmark application after deploying it on all the four cores can be calculated as shown below. From the calculations it can be seen that the sampling frequency obtained in this case is double the sampling frequency obtained in case of dual core and tri core processors.

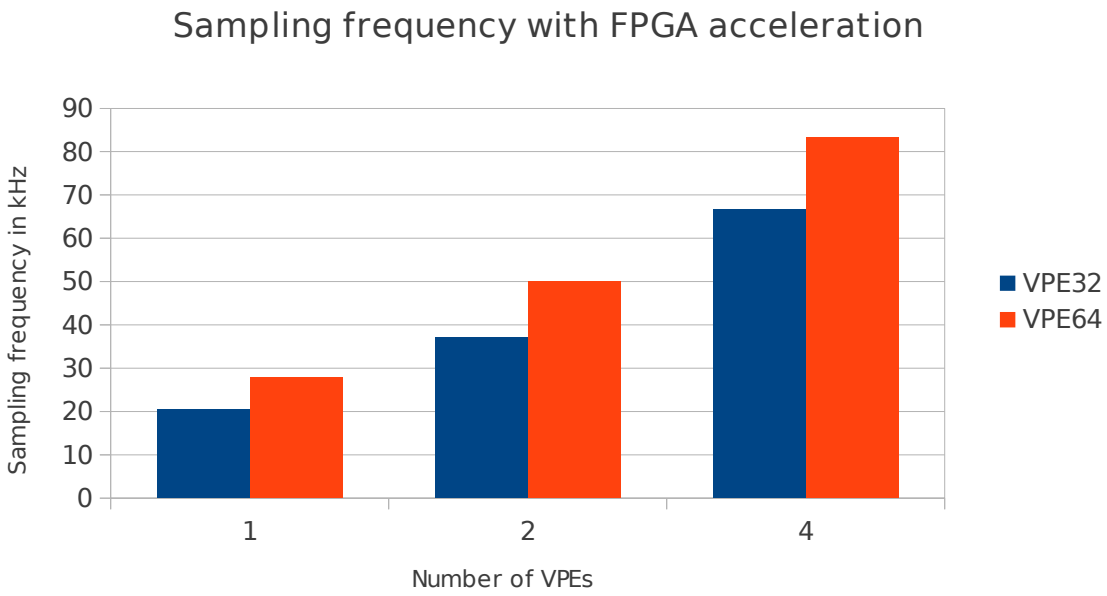

Figure 6.3: Sampling frequency of the benchmark application after deploying the state-space blocks on tri cores, VPE32 and VPE64

$$
\begin{aligned}
ExecTime_{QuadCore} = max(&ExecTime_{SSBlockOn1stCore} + ExecTime_{LongStroke},\\
&ExecTime_{SSBlockOn2ndCore} + (ExecTime_{OtherBlocks})/3,\\
&ExecTime_{SSBlockOn3rdCore} + (ExecTime_{OtherBlocks})/3,\\
&ExecTime_{SSBlockOn4thCore} + (ExecTime_{OtherBlocks}/3))\\
ExecTime_{QuadCore} &= max(153180.97, 154608.16, 154608.16, 154608.16)\\
ExecTime_{QuadCore} &= 154.60\,\mu s = 0.154ms
\end{aligned} \tag{6.14}
$$

$$
SampFreq_{QuadCore} = 1/ExecTime_{QuadCore} = 1/0.154 = 6.49kHz \tag{6.15}
$$

6.5.1.1 32 wide and 64 wide VPEs

Here we consider four cores, VPE32 and VPE64 for experiments. The sampling frequency of the benchmark application obtained when a state-space block is deployed on a VPE32 and VPE64 can be seen in the Table 6.5.

It can be observed from the above table that the hypothesis can be justified when a quad core processor is used in combination with 4 VPE32, 2 VPE64 and 4 VPE64 respectively.

Figure 6.4 shows the sampling frequency of the benchmark application after deploying it on four cores, VPE32 and VPE64.

	No VPE	1 VPE32	2 VPE32	4 VPE32	1 VPE64	2 VPE64	4 VPE64
Max achievable frequency (kHz)	6.49	20.40	37.03	66.67	27.77	50	83.33

Table 6.5: Sampling frequency of the benchmark application after deploying the state-space blocks on a quad core processor and VPEs

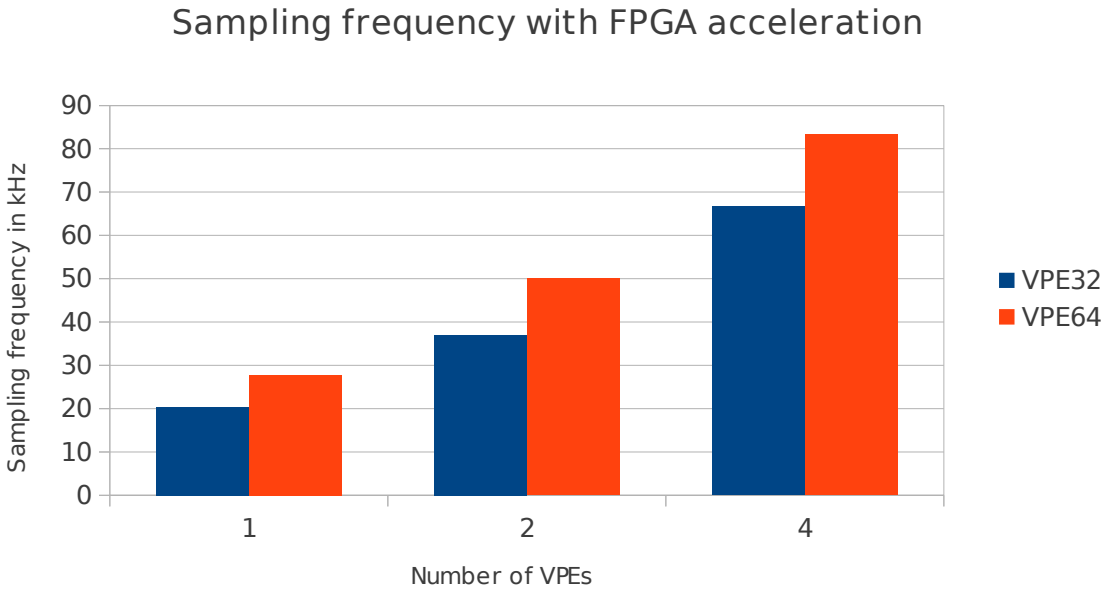

Figure 6.4: Sampling frequency of the benchmark application after deploying the benchmark application on four cores and state-space blocks on VPE32 and VPE64

6.6 Penta-core PowerPC + VPEs

In this section, we consider a penta-core processor and 32 wide and 64 wide VPEs for our experiments. The execution time and sampling frequencies are obtained after deploying the benchmark application on five cores.

6.6.1 Penta-core

The benchmark application is deployed on a penta-core processor. The Long Stroke controller is deployed on one core and the Short Stroke controller is split across four cores. The four state-space blocks present in the Short Stroke controller can now be executed in parallel on the four cores and hence, the execution time of the Short Stroke controller is nothing but the execution time of one state-space block plus some other blocks.

$$ExecTime_{ShortStrokeOnPentaCore} = ExecTime_{ShortStroke}/4$$
$$ExecTime_{ShortStrokeOnPentaCore} = 617,543/4$$
$$ExecTime_{ShortStrokeOnPentaCore} = 154.385\,\mu s = 0.154ms \quad (6.16)$$

The execution time and the sampling frequency of the benchmark application is as shown below. It can be seen from the calculations that the sampling frequency obtained in this case is almost same as that obtained in case of quad core processors because of the below mentioned calculations.

$$ExecTime_{PentaCore} = max(ExecTime_{LongStroke}, ExecTime_{ShortStrokeOnPentaCore})$$
$$ExecTime_{PentaCore} = max(2.479, 154, 385.75)$$
$$ExecTime_{PentaCore} = 154.385\,\mu s = 0.154ms \quad (6.17)$$

$$SampFreq_{PentaCore} = 1/ExecTime_{PentaCore} = 1/0.154 = 6.49kHz \quad (6.18)$$

6.6.1.1 32 wide and 64 wide VPEs

In this experiment, we consider five cores in combination with VPE32 and VPE64. The sampling frequencies obtained when VPE32 and VPE64 are used in combination with a penta core processor are shown in the Table 6.6.

	No VPE	**1 VPE32**	**2 VPE32**	**4 VPE32**	**1 VPE64**	**2 VPE64**	**4 VPE64**
Max achievable frequency (kHz)	6.49	20.40	37.03	66.67	27.77	50	83.33

Table 6.6: Sampling frequency of the benchmark application after deploying the state-space blocks on a penta core processor and VPEs

From the table it can be observed that the hypothesis is justified when 4 VPE32, 2 VPE64 and 4 VPE64 are used in combination with a penta core processor. Figure 6.5 shows the sampling frequencies of the benchmark application when the state-space blocks are deployed on VPE32 and VPE64.

6.7 Octa-core PowerPC + VPEs

The execution time and the sampling frequency of the benchmark application running on an octa-core processor is double the sampling frequency obtained in case of quad core

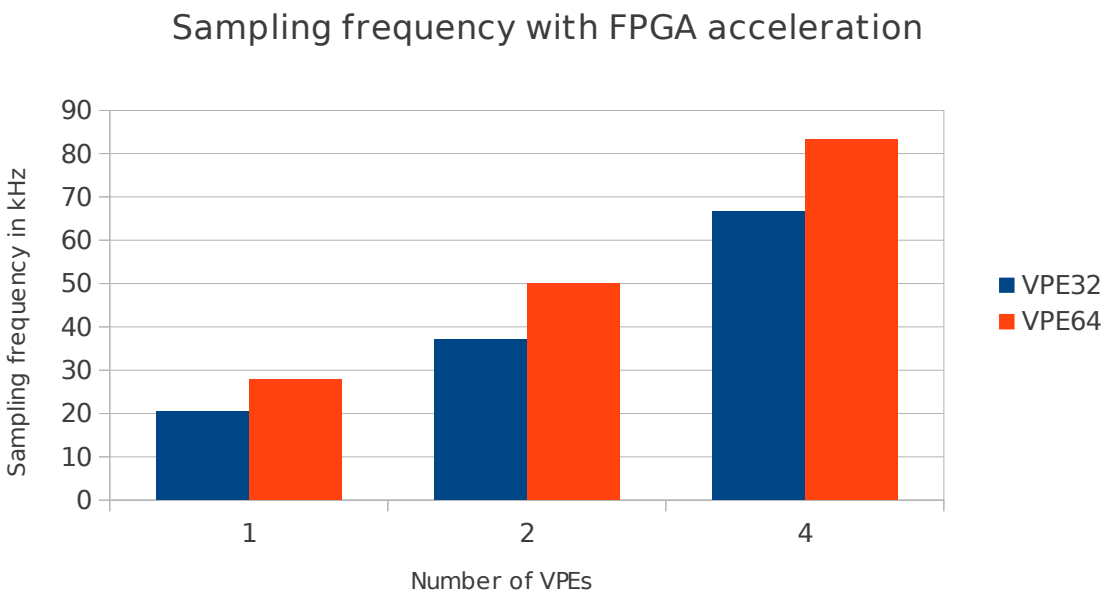

Figure 6.5: Sampling frequency of the benchmark application after deploying the benchmark application on five cores and state-space blocks on VPE32 and VPE64

or penta core processors. This is because the state-space blocks can be executed across eight cores and all the other blocks can be run on 8 cores in parallel. The sampling frequencies obtained in this experiment can be seen in the Table 6.7.

	No VPE	**1 VPE32**	**2 VPE32**	**4 VPE32**	**1 VPE64**	**2 VPE64**	**4 VPE64**
Max achievable frequency (kHz)	13	20.40	37.03	66.67	27.77	50	83.33

Table 6.7: Sampling frequency of the benchmark application after deploying the state-space blocks on a octa core processor and VPEs

From the above table it can be once again observed that the hypothesis is justified by using octa core processor in combination with 4 VPE32, 2 VPE64 and 4 VPE64 respectively. Figures 6.6 and 6.7 shows the sampling frequencies of the benchmark application when the state-space blocks are deployed on VPE32 and VPE64 respectively.

6.8 Experiments by varying number of inputs and by varying number of states

In this section, various experiments are performed by varying the number of inputs and by varying the number of states.

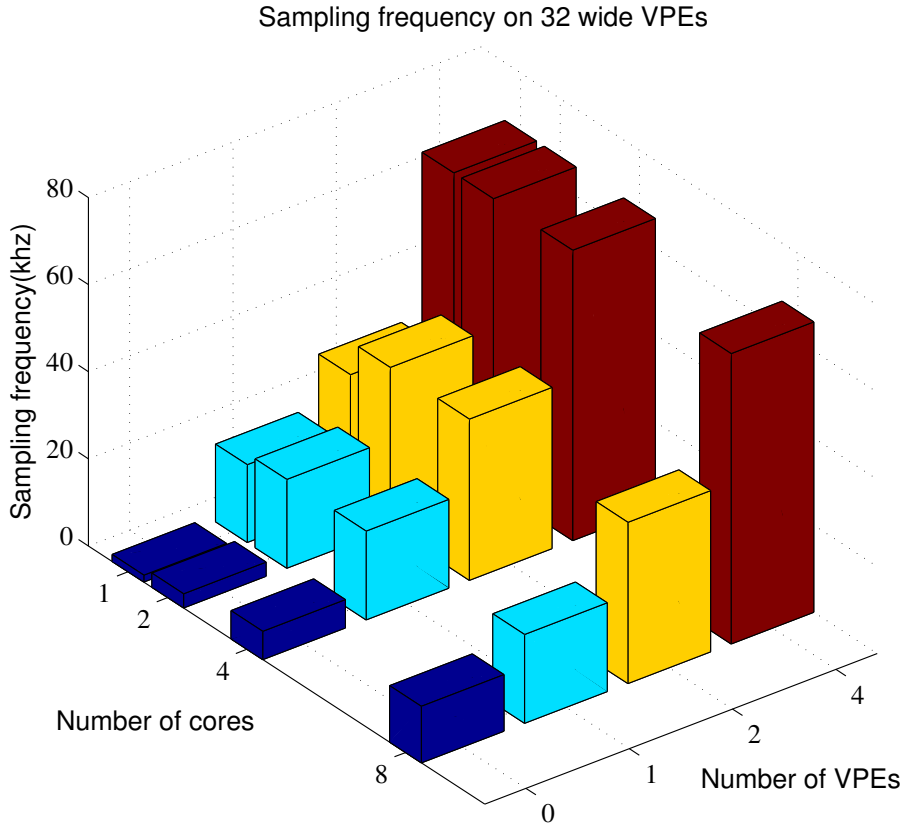

Figure 6.6: Sampling frequency of the benchmark application after deploying the benchmark application on cores and 32 wide VPEs

6.8.1 Varying number of inputs

In this section, we perform experiments by varying the number of inputs and outputs from 32 to 320 in steps of 20, and keeping the number of states constant to 220. The execution time of the state-space block is obtained by executing it on the PowerPC, 32 wide VPE and 64 wide VPE. The purpose of this experiment is to show that, the communication time between the CPU and FPGA plus the execution time on the FPGA, is significantly smaller than the execution time on PowerPC.

Table 6.8 shows the execution time of the state-space block on the PowerPC, 32 wide VPE and 64 wide. Figures 6.8 and 6.9 compares the execution time of state-space block on PowerPC with the execution time on 32 wide VPE and 64 wide VPE. It can be seen from the tables and the figures that, the execution time on FPGA is significantly smaller than the execution time on the PowerPC.

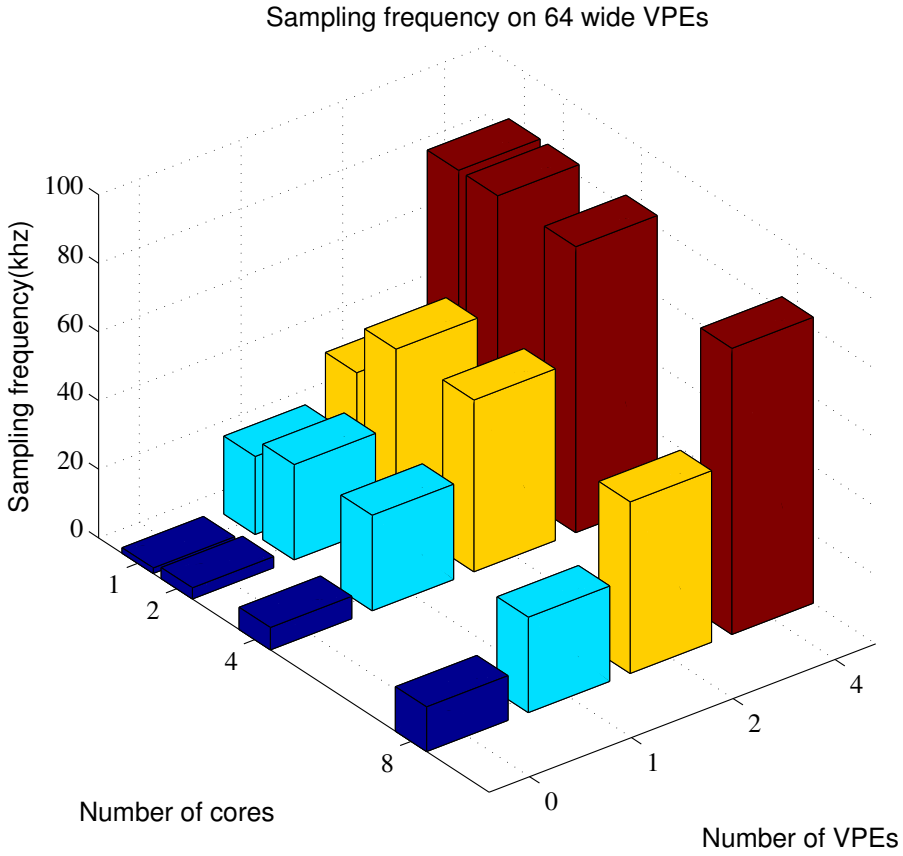

Figure 6.7: Sampling frequency of the benchmark application after deploying the benchmark application on cores and 64 wide VPEs

Number of inputs	**Exec time on PowerPC**	**Exec time on 32 wide VPE**	**Exec time on 64 wide VPE**
32	186.41 μs	15.73 μs	11.99 μs
64	235.4 μs	24.49 μs	16.46 μs
96	284.13 μs	31.4 μs	24.36 μs
128	334.21 μs	38.26 μs	28.69 μs
160	399.55 μs	45.3 μs	34.97 μs
192	560.04 μs	52.16 μs	39.3 μs
224	685.07 μs	59.16 μs	45.53 μs
256	773.07 μs	66.07 μs	49.91 μs
288	986.44 μs	72.92 μs	56 μs
320	1067.75 μs	79.96 μs	60.51 μs

Table 6.8: Execution time of state-space block on PowerPC, 32 wide VPE and 64 wide VPE after varying the number of inputs

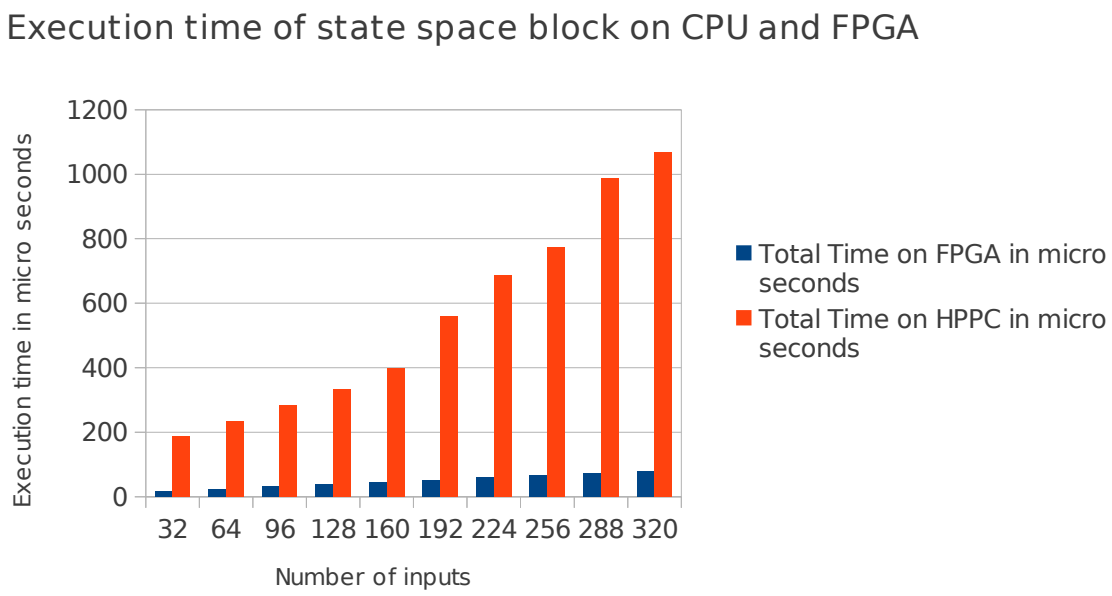

Figure 6.8: Execution time of state-space block on PowerPC and 32 wide VPE after varying the number of inputs

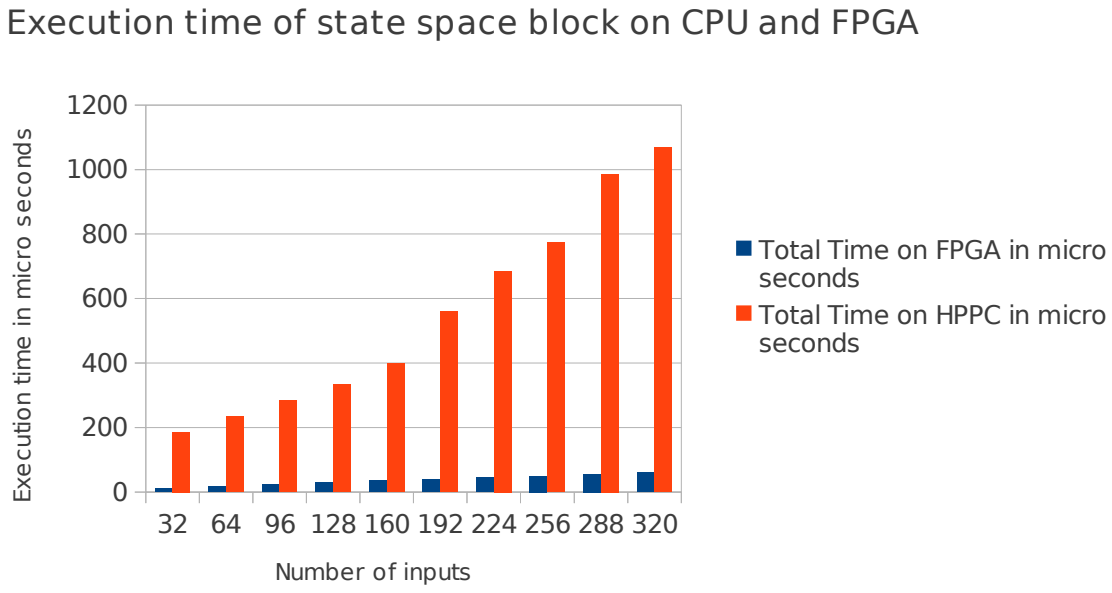

Figure 6.9: Execution time of state-space block on PowerPC and 64 wide VPE after varying the number of inputs

6.8.2 Varying number of states

In this section, we perform the experiments by varying the number of states from 220 to 20 in steps of 20, and keeping the number of inputs and outputs constant to 64. The purpose of this experiment is also to show that, the execution time on FPGA plus the

communication time between CPU and FPGA is smaller than the execution time on PowerPC.

In the Table 6.9 the execution time of the state-space block on PowerPC, 32 wide VPE and 64 wide VPE are shown. Figures 6.10 and 6.11 shows the graph that compares the execution time of state-space block on PowerPC, 32 wide VPE and PowerPC, 64 wide VPE respectively. It can be seen from the table and the figures that, the execution time on FPGA is much smaller than the execution time on the CPU.

Number of states	**Exec time on PowerPC**	**Exec time on 32 wide VPE**	**Exec time on 64 wide VPE**
220	235.4 μs	24.49 μs	16.46 μs
200	202.34 μs	22.73 μs	15.5 μs
180	172.03 μs	20.55 μs	14.12 μs
160	143.84 μs	18.95 μs	13.32 μs
140	117.99 μs	17 μs	12.52 μs
120	94.02 μs	15.11 μs	11.27 μs
100	72.8 μs	13.83 μs	10.63 μs
80	53.62 μs	12.17 μs	9.99 μs
60	36.38 μs	10.57 μs	8.87 μs
40	17.93 μs	9.61 μs	8.39 μs
20	7.75 μs	8.31 μs	8.09 μs

Table 6.9: Execution time of state-space block on CPU, 32 wide VPE and 64 wide VPE after varying the state matrix

6.9 Control mode switching

It has been estimated that the total time taken on an FPGA to load a single parameter set from DDR3 to ASIP memory is 0.2 ms. Since there are four state-space blocks in the benchmark application, the total time taken to load all the four parameter sets from DDR3 to ASIP memory would be 0.8 ms. If we consider 40 kHz as the sampling frequency, the sample time can be calculated as shown below.

$$SampleTime = 1/SampFreq = 1/40000 = 0.025ms \tag{6.19}$$

In the current software architecture, there are 50 samples reserved to load the parameter sets of all the blocks from DDR3 to ASIP memory, and to perform the control mode switch. Hence, the total time required for 50 samples can be calculated as shown below:

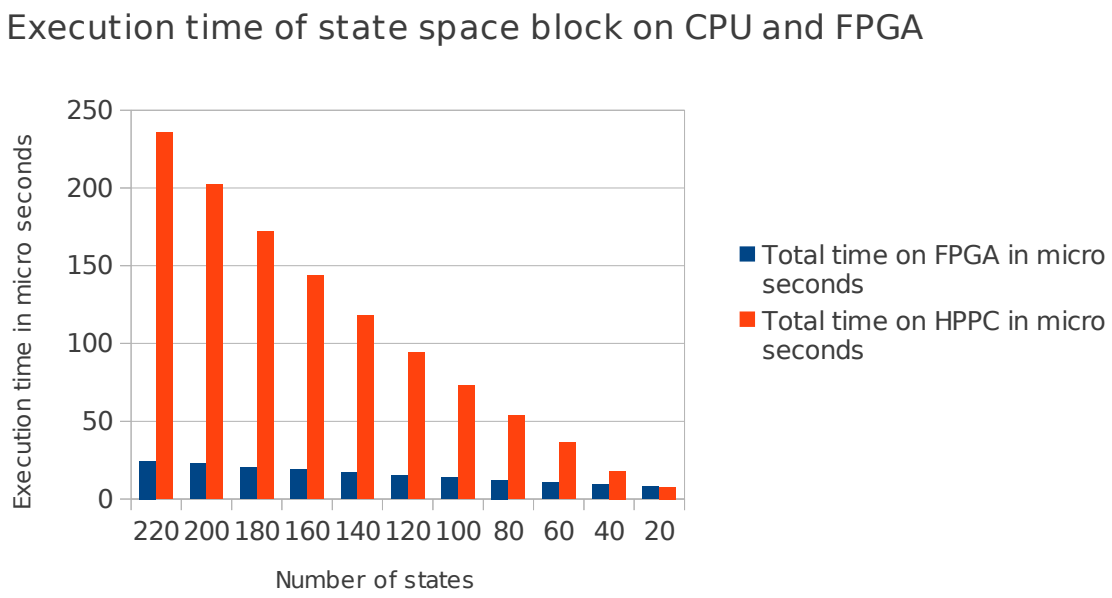

Figure 6.10: Execution time of state-space block on PowerPC and 32 wide VPE after varying the state matrix

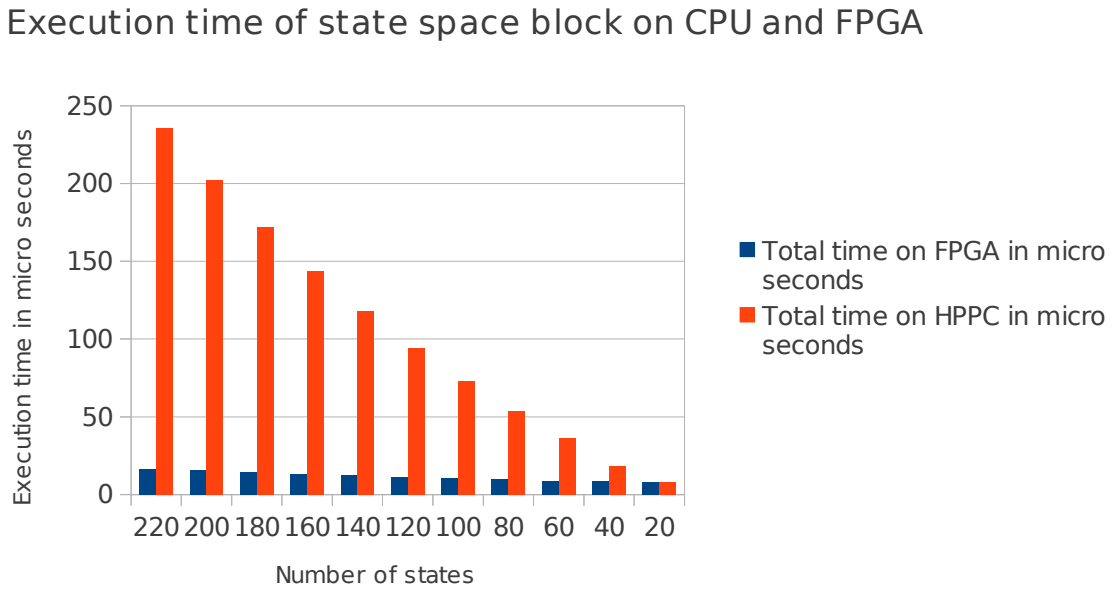

Figure 6.11: Execution time of state-space block on PowerPC and 64 wide VPE after varying the state matrix

$$
\begin{aligned}
TotalTimeFor50Samples &= SampleTime * NoOfSamples \\
TotalTimeFor50Samples &= 0.025 * 50 \\
TotalTimeFor50Samples &= 1.25ms
\end{aligned} \tag{6.20}
$$

Hence, the control mode switch time, i.e., the time taken to send the parameter set load command to the FPGA, loading the parameter sets from DDR3 to ASIP memory on FPGA and sending back the acknowledgement, should be less than 1.25 ms. The control mode switch time can be calculated as shown below:

$$\begin{aligned} CMSTime &= CmdFromCPUToFPGA + ParameterSetLoadTimeOnFPGA + \\ &\qquad AckFromFPGAToCPU \\ CMSTime &= (0.5\,\mu s) + (0.8ms) + (0.5\,\mu s) \\ CMSTime &= 0.8ms \end{aligned} \tag{6.21}$$

It can be seen from the calculations that, the total control mode switch time is less than the total time taken for 50 samples, i.e., 0.8 ms is less than 1.25 ms. However, it should be ensured that the command to load the parameter set to the FPGA, should be given in the first 10 samples out of the 50 samples reserved. Finally, we conclude by saying that, the sampling frequency of 40 kHz can be achieved despite the communication between the CPU and FPGA, and despite the control mode switching functionality.

7 Conclusions and future work

In this chapter, the conclusion is drawn on the work carried out, and present recommendations for further continuation of this work.

7.1 Conclusions

In this thesis, we hypothesized that a sampling frequency of 40 KHz could be achieved, by using an FPGA as an accelerator in conjunction with a CPU. With a detailed analysis, parts of the software architecture, that should be deployed on the hardware platform in order to achieve this projected sampling frequency were identified. Based on our analysis, the hardware platform was selected, that would be best suitable for this project. The analysis carried out also confirmed that the sampling frequency of 40 KHz can be achieved.

Various experiments were performed in order to justify the hypothesis. In the first phase, measurements of all the blocks present in the benchmark application were obtained by executing them on a 1.33 GHz Power PC processor. The execution time of the entire benchmark application obtained was 621.67 μs and the sampling frequency obtained was 1.61 KHz.

In the second phase, a design space exploration was carried out where compute intensive blocks were offloaded from single core, dual core, tri core, quad core, penta core and octa core processors to 32 wide and 64 wide vector processing units (ASIP), on an FPGA. The maximum number of inputs of state-space blocks and the maximum number of outputs of state-space blocks in the current benchmark application is 12. Initially, single core and only one 32 wide vector processing unit was chosen, and 4 state-space blocks present in the benchmark application were deployed sequentially on the VPE. We could observe that the total time obtained (execution time of blocks other than the state-space blocks on CPU + communication time between CPU and FPGA + execution time of state-space block on FPGA) was 61.8 μs, and as a result, the sampling frequency obtained in this case was 17.85 KHz. Then, two and four 32 wide VPEs were chosen respectively, and the sampling frequency obtained were 29.41 KHz and 66.67 KHz respectively. Hence, it was observed that the hypothesis can be justified by considering four VPEs. Similar experiments were performed for the rest of the cores, and it was observed that the hypothesis can be justified by considering four 32 wide VPEs. The experiments were also carried out on 64 wide VPEs, and in this case, we could observe that the hypothesis can be justified two and four 64 wide VPEs respectively.

Various other experiments were carried out in order to prove that, the execution time of the state-space block on the CPU supersedes the communication time between the CPU and FPGA plus the execution time of the state-space block on an FPGA. Intially, the number of inputs and outputs wre varied starting from 32 till 320, in steps of 32, and

keeping the number of states constant to 220. The execution times of the state-space block was obtained for the above mentioned inputs and outputs, by executing them on the Power PC. The time taken to communicate the data with the FPGA was also obtained. We could also get the measurements of the state-space block with the above mentioned inputs and outputs on the FPGA. We observed that, the execution time of the state-space block on the CPU, exceeded the communication time between CPU and FPGA and the execution time of the state-space block on the FPGA, by greater margin. The above conclusion was drawn for both the 32 wide VPE and 64 wide VPE.

Also, the number of inputs and outputs were kept constant to 64, and number of states were varied from 220 to 20, in the steps of 20. The rationale behind carrying out this experiment was to prove that, the execution time of a state-space block on CPU exceeds the communication time between the CPU and FPGA plus the execution time on the FPGA. First, the execution time of the state-space blocks on the CPU were obtained, by varying the number of states, and keeping the number of inputs and outputs constant. Then, the communication time between the CPU and FPGA was calculated, and finally, the execution time of the state-space block on an FPGA was measured, by varying the number of states, and keeping the numnber of inputs and outputs constant. It was observed that, the execution time of the state-space block on the CPU exceeds the communication time between the CPU and FPGA plus the execution time of the state-space block on the FPGA. Based on all the experiments performed, it was concluded that, using FPGA as an accelerator in conjunction with the CPU, benefits by greater margin.

The experiments on control mode switching were also performed, and it was found out that, the control mode switch time on an FPGA plus the communication of parameter set load command between CPU and FPGA was less than the sample time reserved for control mode switching. From this experiment, it can be concluded that control mode switchng can be easily performed on an FPGA. However, the parameter set load command to load the paramter sets from DDR3 to local memory on an ASIP, should be given within the first 10 samples out of the 50 samples reserved. Failure to do so would result in the failure of control mode switching on an FPGA.

7.2 Future work

In this section, we recommend some of the ideas that could be used in the future projects. Following are the ideas that we recommend:

Xilinx Zynq-7000 SOC as a hardware platform: In our thesis, we make use of Xilinx Virtex-7 FPGA as a hardware development kit, as it satisfies all the conditions required for our project. We also thought of considering Xilinx Zynq-7000 SOC [12] as a development platform, as it provides sufficient resources for our project. However, there were other reasons, that compelled us to not to go with Xilinx Zynq-7000 SOC platform. Some of the reasons are as follows: The SOC consists of an ARM architecture along with the FPGA. Since the Wafer Stage motion control application runs on PowerPC processor, porting the entire Wafer Stage to run on an ARM architecture was too big a

step to achieve in the time window of this project. The Zynq-7000 SOC was not available at the start of the project.

For the continuation of this work, we strongly recommend the use of the Zynq-7000 SOC as hardware development platform. This would reduce the communication time between the CPU and FPGA, as both are present on the same chip. We also recommend to compare the performance of our work with that of SOC, and then decide on the best platform out of SOC and FPGA, that could be used in long term in the future.

Deploying more blocks on ASIPs on FPGA: In our project, we carried out experiments by offloading only the state-space blocks. However, there might be other blocks in the benchmark application, that could be accelerated by deploying them on the other ASIPs, such as, Scalar processing units (SPU) and Look-up units (LUs). We recommend the usage of these ASIPs in the near future, as this might further increase the sampling frequency of the entire benchmark application.

Deploying the entire ServoGroup on an FPGA: Currently, in this thesis, FPGA is employed as an accelerator in conjunction with the CPU. In our project, we deploy the blocks present in the benchmark application on an FPGA, however, the other blocks in the benchmark application run on the CPU. The interface of the block that is deployed on an FPGA is present on the CPU, however, the functionality of the block is offfloaded on the FPGA. Hence, the entire control network still remains on the CPU, i.e., the Host can communicate with the FPGA, only through the Worker. In the future projects, we recommend to port the entire control network on an FPGA, so that the Host could communicate with the FPGA directly. This might increase the sampling frequency, as the communication between the CPU and FPGA, would no more be a bottleneck.

Deploying the entire motion control application on an FPGA: Finally, once it is proved that, either an FPGA or an SOC could perform better than the existing hardware architecture, a project can be carried out to port the entire Wafer stage application on an FPGA or an SOC. This could improve the performance of the applications.

Bibliography

[1] D. S. P. Hernandez, "A design-space exploration for high-performance motion control," *ASML internal documentation*, 2012.

[2] R. R. H. Schiffelers, W. Alberts, and J. 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*, ser. MPM '12. New York, NY, USA: ACM, 2012, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/2508443.2508453

[3] N. Gidalov, "Introduction to process control and carm," *ASML internal documentation*, 2012.

[4] T. Kamp, "Heterogeneous motion control processing platform on fpga," *ASML internal documentation*, 2012.

[5] "Dataflow-based multi-asip platform approach for digital control applications," *2013 Euromicro Conference on Digital System Design*, vol. 0, pp. 811–814, 2013.

[6] "Gid carm metamodel definitions," *ASML internal documentation*, 2011.

[7] "Eps pg blocks general," *ASML internal documentation*, 2014.

[8] "Rapidio interconnect specification," 2011.

[9] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[10] G. Martin and H. Chang, "System-on-chip design," in *ASIC, 2001. Proceedings. 4th International Conference on*, 2001, pp. 12–17.

[11] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (mpsoc) technology," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 10, pp. 1701–1713, Oct 2008.

[12] Xilinx, "Zc702 evaluation board for the zynq-7000 xc7z020 all programmable soc," 2012. [Online]. Available: http://www.farnell.com/datasheets/1678513.pdf