Universität Stuttgart

Institute of Computer Architecture and Computer Engineering

University of Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Master's Thesis Nr. 3491

Accelerated Computation using Runtime Partial Reconfiguration

Naresh Ganesh Nayak

Course of Study:	Information Technology/InfoTECH
Examiner:	Prof. Dr. rer. nat. habil.
	Hans-Joachim Wunderlich
Supervisors:	Michael Kochte Dipl.-Inf.
	Michael Imhof Dipl.-Inf.
	Francesco Cervellera M.Sc.
Commenced:	2013-05-27
Completed:	2013-11-26
CR-Classification:	B.5.1, C.4, I.4.9

Acknowledgements

As I present the results of my Masters thesis, I would like to thank the many people who made this success possible.

To begin with, I would like to express my heartfelt gratitude to Prof. Dr. Hans-Joachim Wunderlich for providing me with the opportunity to contribute towards the research project, OTERA, with this thesis. The frequent suggestions and reviews provided by him were critical for the success of this thesis. I would like to thank my thesis supervisors - Mr. Michael Kochte, Mr. Michael Imhof and Mr. Francesco Cervellera. They took keen interest in my work and motivated me all along to meet the goals of the thesis. I feel lucky to have been supervised by them. Without their invaluable guidance and direction, I would have been surely lost.

I would like to specially thank the system administrators of the department — Mr. Helmut Häfner and Mr. Lothar Hellmeier — for providing all the necessary infrastructure for the thesis expeditiously. I am also thankful to all the staff members of the institute for making my thesis at the institute an enjoyable experience.

Finally, I would like to thank all my friends and colleagues who were with me during all the ups and downs of the Masters thesis. Going through all of this alone would have been difficult.

Abstract

Runtime reconfigurable architectures, which integrate a hard processor core along with a reconfigurable fabric on a single device, allow to accelerate a computation by means of hardware accelerators implemented in the reconfigurable fabric. Runtime partial reconfiguration provides the flexibility to dynamically change these hardware accelerators to adapt the computing capacity of the system. This thesis presents the evaluation of design paradigms which exploit partial reconfiguration to implement compute intensive applications on such runtime reconfigurable architectures. For this purpose, image processing applications are implemented on Zynq-7000, a System on a Chip (SoC) from Xilinx Inc. which integrates an ARM Cortex A9 with a reconfigurable fabric.

This thesis studies different image processing applications to select suitable candidates that benefit if implemented on the above mentioned class of reconfigurable architectures using runtime partial reconfiguration. Different Intellectual Property (IP) cores for executing basic image operations are generated using high level synthesis for the implementation. A software based scheduler, executed in the Linux environment running on the ARM core, is responsible for implementing the image processing application by means of loading appropriate IP cores into the reconfigurable fabric. The implementation is evaluated to measure the application speed up, resource savings, power savings and the delay on account of partial reconfiguration.

The results of the thesis suggest that the use of partial reconfiguration to implement an application provides FPGA resource savings. The extent of resource savings depend on the granularity of the operations into which the application is decomposed. The thesis could also establish that runtime partial reconfiguration can be used to accelerate the computations in reconfigurable architectures with processor core like the Zynq-7000 platform. The achieved computational speed-up depends on factors like the number of hardware accelerators used for the computation and the used reconfiguration schedule. The thesis also highlights the power savings that may be achieved by executing computations in the reconfigurable fabric instead of the processor core.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

Ever since their invention two decades back, reconfigurable architectures have been evolving continuously. Their popularity can be associated to the fact that they combine the flexibility of software with the performance of hardware. Field Programmable Gate Arrays (FPGAs) are perhaps the most popular and widely used class of reconfigurable architectures. With the research community devoting significant resources for developing these architectures, it is hardly surprising that they are growing larger and becoming faster by the day. Xilinx Inc. and Altera Corporation, world's leading vendors of FPGAs, have been introducing larger and faster products with every passing generation along with additional features which enhance their usability. In their recent product releases, they have introduced the feature of *Partial Reconfiguration* which allows the modification of selected FPGA areas while the logic in the remaining area continues to function. This ability of an FPGA to modify the implemented logic during operation takes its flexibility one step further [1].

The concept of partial reconfiguration of FPGAs has been around in academia for long, with lot of research on exploring different use cases where the additional flexibility can be put to effective use. Publications proposing usage of partial reconfiguration for purposes varying from power efficient designs to improving fault tolerance have been published by the scientific community [2, 3, 4, 5]. Computation platforms utilising partial reconfiguration with different granularities have been presented in [6, 7]. Applications like video processing implemented using partial reconfiguration have been introduced in [8]. Despite all the work in developing

partial reconfiguration as a design methodology, its usage is limited to research applications. This can be attributed to the lack of design tools for efficient implementation and commercial applications which may significantly benefit from it to justify the effort required for its complex design flow [9, 10]. To promote the usage of partial reconfiguration in commercial applications, FPGA vendors have made available reference designs to exhibit its potential [11].

This thesis explores the technical details underlying the process of partial reconfiguration in FPGAs and implements compute intensive applications on runtime reconfigurable architectures with a processor core using partial reconfiguration. The thesis further evaluates the implementation to quantify the benefits and overheads involved in such designs.

This thesis lies within the framework of the project *Online Testing Strategies for Reliable Reconfigurable Architectures* (OTERA) funded by the *Deutsche Forschungsgemeinschaft* (DFG) [12]. The primary goal of this project is enhancing the reliability of reconfigurable architectures which use runtime partial reconfiguration by guaranteeing the health of the reconfigurable fabric. This project focuses on developing online test approaches which monitor and verify reliable functioning of the underlying reconfigurable fabric by means of periodic tests [13, 14]. The partial reconfiguration based implementation developed as a part of the thesis aims to be re-usable for the OTERA project for validating the testing approaches researched within the project.

1.2 Goals

1.2.1 Study of Partial Reconfiguration in FPGAs

The thesis commences with the study of the concept of partial reconfiguration, its underlying technical challenges and the relevant publications in the field. Design tool chains which support partial reconfiguration based design have also been explored. While there is no bias towards any particular FPGA vendor, tool chains and FPGAs from Xilinx Inc. are primarily investigated, mainly because of its synergy with other implementations within the OTERA project [15, 16].

Partial reconfiguration is a method which needs to be used appropriately to unlock its potential. Researchers and design community have put forth certain design patterns which utilises partial reconfiguration as an effective tool. Different ways in which partial reconfiguration can be used effectively have been presented in [1]. New design techniques use partial reconfiguration to implement novel features which would have been otherwise impossible. As a part of the thesis, different design paradigms which use partial reconfiguration have been explored and summarised.

1.2.2 Study of Image Processing Applications

One of the primary goals of the thesis is to implement compute intensive applications on a reconfigurable architecture using partial reconfiguration. The choice of the application(s) to

(a) Camera on-board a car (b) Lane Detection with Image Processing

Figure 1.1: Use case for Image Processing Algorithms - Lane Detection

be implemented is critical for deriving meaningful results from the thesis. Applications which have static resource requirements rarely tend to benefit from partial reconfiguration. As a part of the thesis, different applications are investigated for their suitability to be implemented with partial reconfiguration.

The applications investigated are limited to the domain of image and video processing. Image and video processing algorithms are ubiquitously used these days with most applications using it in some form or other. For instance, Figure 1.1 shows the usage of image processing to detect lanes on a highway, which can be used for warning drivers against accidental lane departures[1]. The choice of image and video processing algorithms for implementation is not only driven by its relevance in current commercial applications, but also by the fact that the complexity of these algorithms is sufficient enough to derive quality data out of the exercise. The abundance of literature and open source implementations available for use in this domain eases the mapping of these algorithms on to reconfigurable architectures [17, 18].

1.2.3 Implementation using Partial Reconfiguration

Applications, suitable to be implemented using partial reconfiguration, are implemented on a runtime reconfigurable architecture with a processor core. The thesis utilises the latest available tool chains for design purposes and attempts to keep the design generic enough for implementing any image processing application. One of the optional goals for the thesis, is to utilise Direct Memory Access (DMA) and interrupts for improving the performance of the implemented applications.

It is important to note that the thesis does not deal with development of customised hardware architectures for image processing, rather it seeks to evaluate the usage of partial reconfiguration for the purpose. Hence, the thesis does not attempt to develop any image processing

[1] Image from Wikipedia - Lane departure warning system, licensed for usage under Creative Commons License

architectures from scratch. Instead this thesis explores usage of open source Intellectual Property (IP) cores for image processing functionalities and high level synthesis tools for generating the required IP cores.

1.2.4 Evaluation of Implementations using Partial Reconfiguration

The final phase of the thesis evaluates the performance of the implemented applications compared to its software and non partial reconfiguration based hardware implementations. The evaluation quantifies the FPGA resource savings, power savings, the time required for performing reconfigurations and the achieved application speed-up, when partial reconfiguration is deployed.

1.3 Thesis Organisation

The organisation of the remainder of this thesis is presented in this section. Chapter 2 summarises the different publications in the field of partial configuration of FPGAs, mainly with respect to its benefits and involved overheads. The different popular platforms and novel FPGA designs based on partial reconfiguration are also briefly mentioned in this chapter.

Chapter 3 explores the different image processing algorithms which can be accelerated using FPGAs. Few applications which can potentially benefit from partial reconfiguration are elaborately explained in the chapter. This chapter also enumerates open source implementations of these image processing algorithms, which can be used as a starting point for an implementation of partial reconfiguration based designs.

Chapter 4 covers the design choices involved in implementing image processing applications on a reconfigurable architecture with a processor core. The involved tool flow is briefly described in this chapter. Chapter 5 describes the implementation of all the selected image processing applications using partial reconfiguration.

Chapter 6 puts forth the result of all the evaluations performed on the implemented image processing applications. Chapter 7 concludes the thesis with summary and possible future work originating from this thesis.

Summary

This thesis explores different design patterns for efficient use of partial reconfiguration. The goals of the thesis include investigating different technical aspects of partial reconfiguration, relevant design tools and applications which can benefit from partial reconfiguration. The thesis also aims to implement compute intensive and industrially relevant applications on a runtime reconfigurable architecture with a processor core for evaluation.

Chapter 2

Partial Reconfiguration in FPGAs

Contents

Partial reconfiguration of a Field Programmable Gate Array (FPGA) deals with modifying an already programmed hardware design in an FPGA. The reconfigurable fabric inside an FPGA is configured by downloading bitstreams generated by synthesis of hardware descriptions. During the configuration of an FPGA, the entire fabric is programmed by use of full bitstreams. With partial reconfiguration, it is possible to program only selected regions of the reconfigurable fabric. This is done by the usage of partial bitstreams which do not contain information for the entire reconfigurable fabric, but only for those regions which are to be (re)configured.

For partial reconfiguration, the hardware design must be divided into two parts - static logic and reconfigurable logic. During the process of partial reconfiguration, only those areas of FPGA are modified where the reconfigurable logic is implemented. The process of reconfiguration may be performed at runtime, i.e. the static logic continues to function even as the reconfigurable logic is undergoing reconfiguration.

As shown in the Figure 2.1, FPGA is configured initially with a complete bitstream, which programs both the static as well as the reconfigurable logic. During subsequent reconfigurations, partial bit streams are loaded which modify only the reconfigurable logic, while not affecting the integrity of the static logic [1].

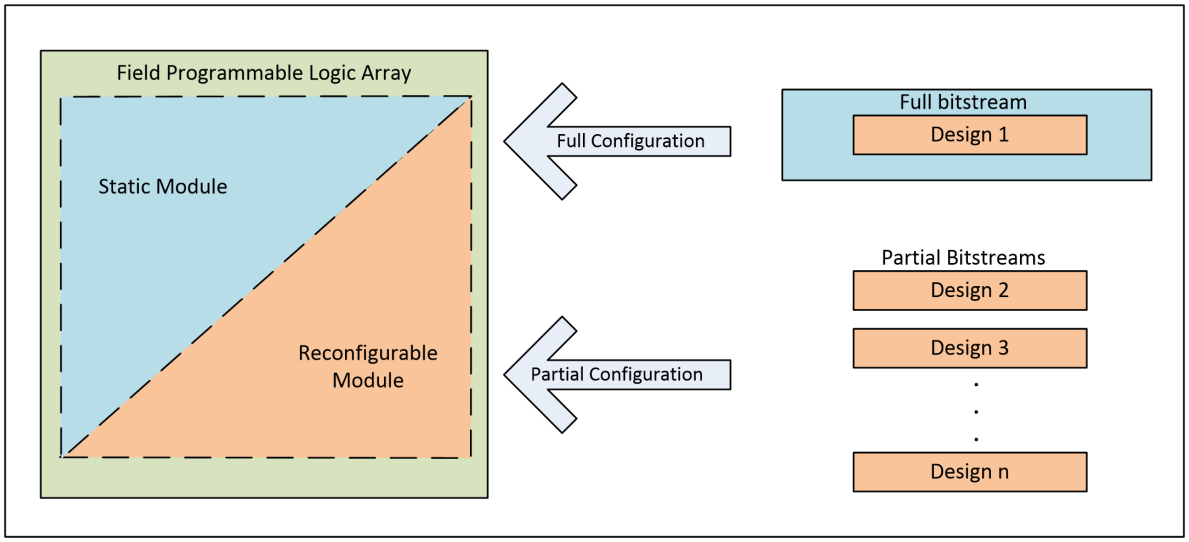


Figure 2.1: Partial Reconfiguration in FPGAs

2.1 Benefits of Partial Reconfiguration

The use of partial reconfiguration opens up a whole new world of FPGA based designs which would have been otherwise difficult to implement. The benefits of using partial reconfiguration include, but are not limited to reducing power consumption and resource utilisation. This section discusses all the benefits of using partial reconfiguration in FPGAs.

Reduced Resource and Power Consumption With partial reconfiguration, it is possible to time multiplex resources between different hardware modules. This reduces the total resource requirement for implementing any hardware design enabling usage of FPGAs with reconfigurable fabrics of smaller size, which translates into power and cost savings. For systems using multiple FPGAs, partial reconfiguration provides the possibility of integrating the design into a lower number of FPGAs. For such systems, power savings can be obtained not only from the reduced number of FPGAs but also from the reduction in off-chip communication [19, 20].

Performance Improvements and Flexibility Runtime partial reconfiguration helps improve the performance of applications by exploiting higher levels of parallelism. With partial reconfiguration, the computation capacity of the system can be adapted at run time. For instance in systems where different hardware accelerators are used sequentially one after the other, runtime partial reconfiguration provides performance improvements by allocating more resources per accelerator. The additional resources can be used for speeding up the operation of the accelerator or for creating more number of accelerators to perform the operation in parallel.

Improved Fault Tolerance and Dependability Fault tolerance is a highly important criterion for safety critical systems. System failures on account of hardware faults could be

fatal in such systems. With partial reconfiguration, fault tolerance and dependability of the systems can be improved by means of techniques like module diversification, configuration scrubbing [2, 4, 21, 22].

Faster System Boot up Just as Moore's law predicted, the number of transistors on an Integrated Chip (IC) has kept doubling biennially. From the perspective of FPGAs, this has resulted in larger reconfigurable fabrics. The growing size of reconfigurable fabrics implies larger bitstreams and hence longer configuration times. The size of full bitstreams for the largest of the current state-of-art 7-series FPGAs from Xilinx Inc. is over 50 MB [23]. Consequently, initial configuration of large FPGAs consumes boot times in the order of hundreds of milliseconds. Many applications cannot tolerate such high start up times and hence cannot use such large FPGAs. For many others, high start up times do not allow fully powering down the FPGA when idle.

Partial bitstreams are smaller and hence require lower configuration times. Start up times can be reduced by loading only partial design consisting of critical components instead of the entire design at start up. Subsequently the remaining part can be loaded using partial reconfiguration after the system has started up. The time required for booting now depends on the size of the partial bitstream which is loaded initially [20].

Self Adapting Hardware Designs Use of partial reconfiguration enables implementation of hardware architectures which are able to adapt themselves to changing operating and environmental conditions. This allows implementation of systems based on artificial intelligence and learning [24].

2.2 Types of Partial Reconfiguration

Partial reconfiguration can be classified based on when the process of reconfiguration is performed [20].

1. Partial reconfiguration performed when the device is not active, for instance by powering off the device or by disabling all the clocks in the design is known as **Passive Partial Reconfiguration**. It is also known as static partial reconfiguration. Passive reconfiguration is supported in Xilinx *Spartan-3*. Passive partial reconfiguration is useful in cases where there is a need to upgrade hardware remotely and system down time is acceptable.
2. Partial reconfiguration performed when device is operational is known as **Active Partial Reconfiguration**. It is also known as dynamic or runtime partial reconfiguration. Xilinx *Virtex* families support active partial reconfiguration. Active partial reconfiguration is useful for implementing self adapting hardware designs.

Xilinx Inc. classifies partial reconfiguration based on the design flow [19, 25].

1. **Difference based Partial Reconfiguration** — Such a partial reconfiguration flow is used for making small changes in the design operating on the FPGA. This design flow configures only the differences between the operating design and the new one.

2. **Module based Partial Reconfiguration** — Module based partial reconfiguration is recommended when large logic blocks are to be reconfigured. Just as the name suggests, this method follows a modular design approach.

2.2.1 Difference based Flow

Of these two partial reconfiguration design flows, the difference based flow works at a lower abstraction level and hence requires an understanding of logic implementation on FPGAs. This design flow is based on the principle of exploiting the correlation between bitstreams of similar designs. This flow is recommended when the reconfiguration is done for achieving minor changes. Such changes include but are not limited to modification of LUT equations, I/O ports and contents of block RAM.

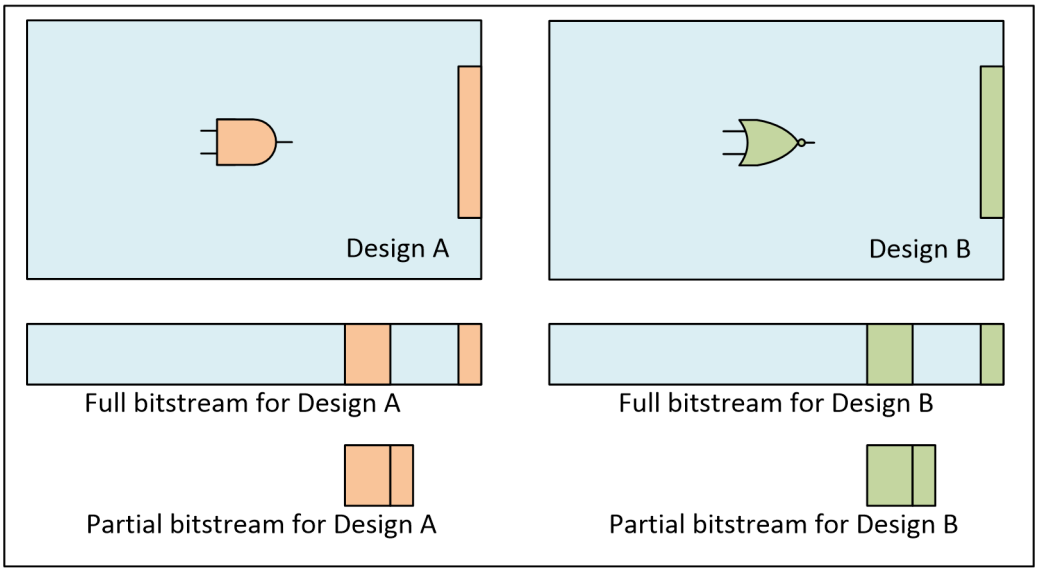


Figure 2.2: Difference Based Partial Reconfiguration

Consider two hardware designs resembling each other, say Design A and Design B. Figure 2.2 shows the difference between the designs — a logic gate and few I/O ports. To configure an FPGA for either of the designs, the full bitstreams which are generated from the synthesis of the corresponding designs are required. If the device requires to switch between the two designs, then during each reconfiguration corresponding full bitstream must be uploaded to FPGA. Instead the full bitstream can be generated for only one of the designs, say Design A. For Design B, a partial bitstream can be created based on the difference between the two designs. This partial bitstream modifies only those regions of the FPGA which are different between Design A and B, hence modifying Design A to get Design B. It is obvious that for small changes the size of partial bitstreams will be much smaller than the full bitstreams. It is important to note that loading this partial bitstream will result in Design B only if Design A is already operating on the FPGA.

In principle, this style of partial bit stream creation is advantageous only if the layouts of the designs to be reconfigured resemble each other. This design flow is practically feasible only for small designs as there is no automated way of creating similar layouts for large unrelated designs [26]. The distinction between static and reconfigurable logic is rather ambiguous in this design flow, especially when more than two designs are considered. In case of multiple designs, logic which is static between two designs might vary with other designs in the application.

The difference based design flow is used for reconfiguring a particular design to another. The design may not be reverted back with the same bitstream. Applications with more designs require more partial bitstreams and hence are more difficult to manage. For instance, application with N designs requiring that any design be reconfigurable to any other arbitrary design requires $N(N-1)$ partial bitstreams. This issue can be mitigated to an extent by fixing one of the designs as a neutral design and reconfiguring the device to this design before loading the desired design. This reduces the required number of partial bitstreams to $2N$ and hence the required storage area for the bitstreams but at the cost of two reconfigurations instead of one. While this approach solves the problem of using many partial bitstreams, it is not always possible to find a neutral design which resembles all the other used designs [26].

Xilinx tool chains support difference based partial reconfiguration for its FPGAs with *FPGA Editor* for modifying LUT equations and I/O Standards and *Data2MEM* for modifying block RAM contents. The reference designs using difference based partial reconfiguration are presented in [27].

2.2.2 Module based Flow

The module based design flow is highly recommended for complex designs where large blocks of logic are to be reconfigured. This design flow exploits the advantage of having common logical blocks between multiple designs. The common logic between such designs is implemented as static logic while the other hardware modules are time multiplexed on the reconfigurable fabric. While the difference based design flow makes small changes inside hardware modules, module based design flow modifies the hardware modules itself. Unlike in difference based design flow, this design flow demands clear demarcation between the static and the reconfigurable logic. The design in Figure 2.3 clearly defines the regions of the FPGA as static or reconfigurable. Hardware modules are loaded into these reconfigurable regions based on a reconfiguration schedule.

This design flow is based on Bottom-Up synthesis, where each hardware module is synthesized independently. Reconfigurable logic is separated from the static logic by means of partitions (also known as containers) — logical sections of the design which are marked for reconfiguration in the top level module of hardware. Partial reconfiguration is achieved by creating design configurations which assign the synthesized hardware modules, designated as reconfigurable modules, to these partitions. Different design configurations assign different hardware modules to each partition and then create partial bitstreams corresponding to each hardware module. Required hardware modules can be loaded into the partitions of the operating design by downloading the corresponding partial bitstreams into the FPGA. As a rule of thumb,

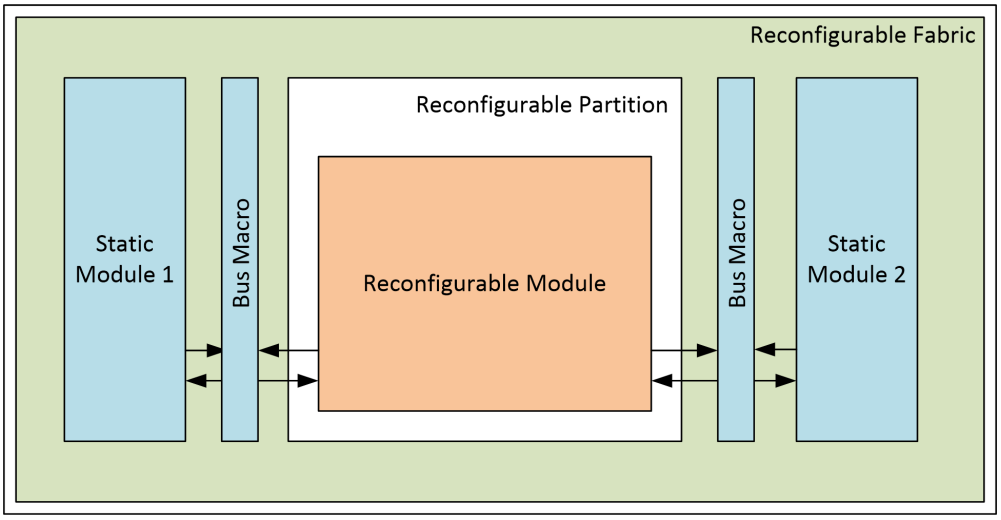
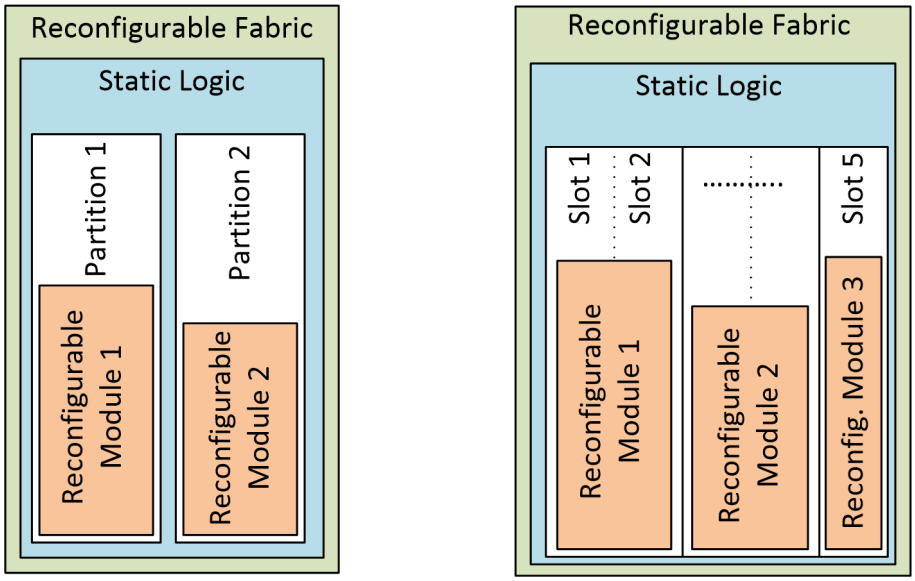


Figure 2.3: Module Based Partial Reconfiguration [27].

the size of the partition should be at least 20% bigger than the size of the biggest hardware module being assigned to it to account for routing overheads [11].

Thus, the module based partial reconfiguration is simply implementing several hardware modules and then time multiplexing the synthesized modules using partitions. Using multiple partitions in a design brings with it the challenge of placing them appropriately in the reconfigurable fabric. Figure 2.4 shows different possible styles of placing reconfigurable modules in a partition [20].

1. **Island Style** - This style allows placing one reconfigurable module per partition — there can be many partitions in a design. Island style of placing the reconfigurable modules can be classified as;
   (a) Single Island Style - The sets of reconfigurable modules for each of the partition are distinct, i.e. the reconfigurable modules can be loaded on only a specific partition and
   (b) Multi Island Style - Reconfigurable modules may be loaded in different partitions.
   As the size of the partition cannot be modified after synthesis and implementation, it must be big enough to allow loading of the largest of the allocated reconfigurable module. This scheme suffers from the problem of internal fragmentation, i.e. wastage of resource on the fabric when smaller modules are loaded into the partition.
2. **Slot Style** - This style calls for division of a partition into multiple tiles along a single dimension (columns, since resources are arranged in columns on the reconfigurable fabric) called slots. The reconfigurable modules are loaded on multiple of these slots based on their resource requirements. This allows loading of multiple reconfigurable modules into a partition, reducing the impact of internal fragmentation like in Island style. However, this style is affected by external fragmentation, i.e. though there are free slots for loading a module, they do not form a continuous area hence preventing the loading of the module. Analogous to memory de-fragmentation in the heap memories of computer systems, compacting can reduce the external fragmentation in this style.



(a) Island Style (b) Slot Style

Figure 2.4: Different styles of placing Reconfigurable Modules in Partitions

The slot style of placing reconfigurable modules increases the design complexity, especially for providing communication to and from the modules.

3. **Grid Style** - Grid style further extends the slot style by dividing the partition into tiles along both dimensions. This reduces the amount of unused resources in the partition, i.e. external fragmentation. Placing the reconfigurable modules in the tiles optimally, wasting least FPGA area, using the grid or the slot style is analogous to the bin packing problem which is *np*-hard.

Using the module based design flow for partial reconfiguration poses numerous challenges for designers. Communication between modules which use bus macros passing through partitions marked for reconfiguration leads to problems, as during reconfiguration there can be no communication between them. Ideally, no communication flows should pass through partitions meant to be reconfigured. Communication between the modules in static logic and reconfigurable logic must be handled via bus macros in static region, like in Figure 2.3. Even then designs must have safety features to prevent any attempt to communicate with modules which are undergoing reconfiguration.

Unlike the partial bitstreams generated using difference based design flow which program only the difference between two designs, the partial bitstreams generated using this flow configure the entire reconfigurable partition for which they are generated. Hence loading a partial bitstream will program the corresponding partition irrespective of the current contents of the partition. This implies, for any application with N design configurations, only N bitstreams are required for changing any arbitrary configuration to any other configuration. However, a

full bitstream should have been loaded at least once for loading the static logic in the fabric following which it cannot be modified again without a full configuration.

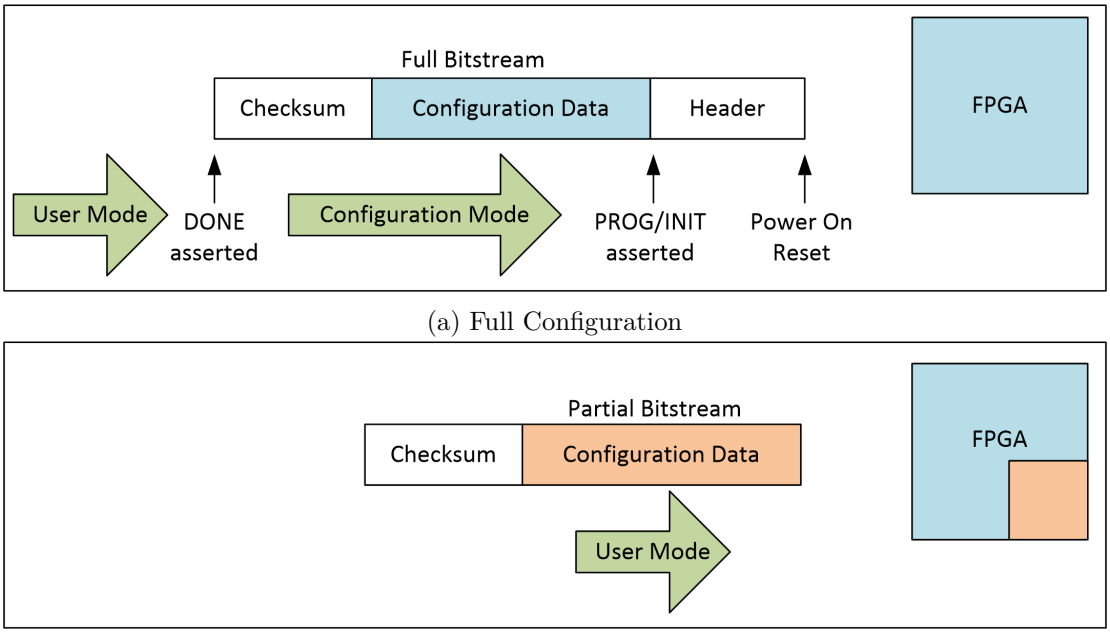
The current versions of the commercial design tools do not support all the advancements with respect to this design flow, owing to the involved complexities in placement and routing. Most of the tools support only the island style of placing the modules. Multi-island style is supported by generation of different partial bitstreams for a hardware module, each associated with a particular partition.

2.3 Process of Partial Reconfiguration

As shown in Figure 2.5, the process of partial reconfiguration is slightly different from the full configuration process, owing to the difference in contents of the partial and full bitstreams. A full bitstream consists of a header, configuration data and checksum. Together they contain all the data necessary to verify the integrity of the bitstream and configure the complete FPGA fabric with the design. At the end of the configuration process, the FPGA asserts the *DONE* signal and enters the user mode from the configuration mode, i.e. the design starts functioning, if the bitstream was not found corrupted. This ensures incorrect designs never start operating on the FPGA. Partial bitstreams, on the other hand, contain only configuration data and checksum. In contrast to the full configuration, the FPGA is already in user mode with an operating design when the reconfiguration process takes place. Hence, asserting the DONE signal is meaningless. The reconfiguration process lasts till the partial bitstream is entirely sent to the configuration port. During the reconfiguration the other areas of the FPGA should not initiate any communication with the areas under reconfiguration.

In order to download a bitstream to a FPGA, it is directed to any of the available configuration ports. Configuration ports are responsible for configuring the FPGA fabric after verifying the integrity of the bitstreams. The different configuration ports generally used are discussed in Section 2.4. Generally, a configuration port has an interface which includes an *INIT*/*PROG* signal for starting the configuration mode. For this purpose, the *PROG* or the *INIT* signal is asserted while downloading a full bitstream. This is not done while downloading partial bitstreams, as the reconfiguration process is carried out in user mode.

Checksum failures indicate corrupt bitstreams and are a cause of concern when detected in partial bitstreams. The checksum, while using partial bitstreams, can be verified only after the entire bitstream has been loaded. By then it is too late, as the incorrect design starts operating on the FPGA. If the checksum failure is caused due to erroneous configuration data, the incorrect design is isolated within the reconfigurable partition and can be corrected by loading another partial bitstream for the corresponding partition. However, if the the failure is caused due to erroneous frame address of the reconfigurable partition, then the static logic may be corrupted as a result of the reconfiguration. To avoid such mishaps especially in systems where bitstreams are transmitted through noisy channels, their integrity needs to be checked before directing them to configuration ports [1].



(a) Full Configuration

(b) Partial Reconfiguration

Figure 2.5: Differences between full configuration and partial reconfiguration [1]

2.4 Configuration Ports

Generally, a processor or a state machine is used for fetching the partial bitstreams from non-volatile memory and directing it to configuration ports for reconfiguration. Different configuration ports can be used for performing partial reconfiguration of an FPGA. A short list of these configuration ports have been presented in this section.

- Internal Configuration Access Port (ICAP)—This port enables partial reconfiguration within the FPGA, thus allowing self configuring FPGA designs. Self configuring FPGA designs with Xilinx Spartan III families have been presented in [28]. Designs for improving the fault tolerance of the ICAP has been presented in [29].
- Processor Configuration Port—This configuration interface is used by the runtime reconfigurable architectures which integrate a hard processor core. The processor configures the reconfigurable fabric using this port. Processor Configuration Access Port (PCAP) in the Zynq-7000 platform from Xilinx Inc. and FPGA Configuration Manager in Arria V devices from Altera Corp. are examples of such ports [30, 31].
- JTAG Port—This is an interface for quick testing of partial reconfiguration. Processors can be used for fetching the bitstreams from the memory and directing it towards the JTAG port. Different tools are available for this purpose [1].

The Figure 2.6 shows the usage of the three mentioned configuration ports. In addition to these ports, serial ports can also be used for the purpose of reconfiguration [1, 31].

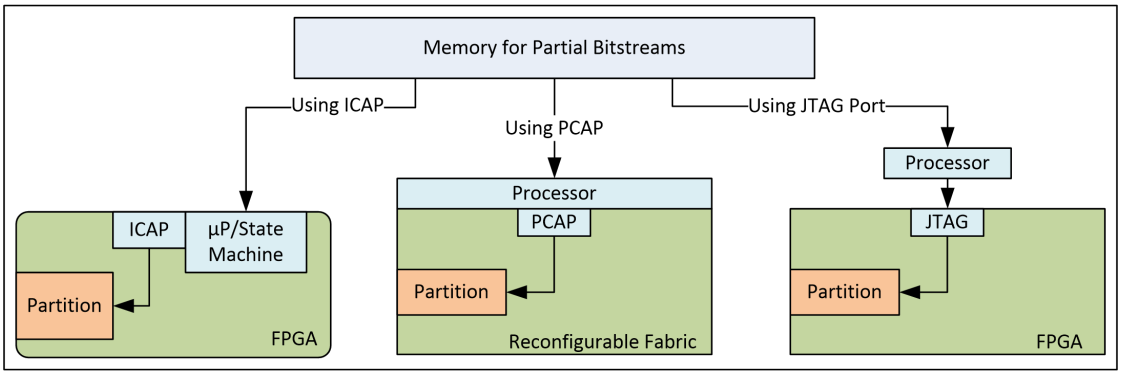


Figure 2.6: Partial Reconfiguration using ICAP, PCAP and JTAG Port [1]

2.5 Tools used for Partial Reconfiguration

The design tools which can be used for creation of hardware designs based on partial reconfiguration depend upon the FPGA in use.

Vendors of FPGAs — Xilinx Inc. and Altera Corporation — have modified their tool suites for allowing designers to employ partial reconfiguration. PlanAhead from Xilinx Inc. and Quartus from Altera Corporation allow users to create designs based on module based partial reconfiguration. PlanAhead allows placing reconfigurable modules in Island style — single as well as multiple. FPGA Editor allows users to design using difference based partial reconfiguration. These tool chains are proprietary and allow only licensed usage.

There are other open source design tools from the research community for implementing partial reconfiguration based FPGA designs. OpenPR, an open source partial reconfiguration toolkit, provides functionality similar to Xilinx tools but is open source and is extendable for accommodating research developments. This toolkit is targeted towards Xilinx Virtex-4 and Virtex-5 architectures but is flexible to accommodate other architectures as well [32]. GoAhead is an efficient tool for using partial reconfiguration on recent Xilinx FPGA products including Spartan-6 series [33].

All these mentioned tools provide support for implementing FPGA based designs using partial reconfiguration. Most of these tools allow design verification by individually testing each valid configuration as they do not support simulation of the reconfiguration process. ReSim[1] is a tool developed for simulation of the reconfiguration process and complete verification of systems deploying partial reconfiguration. Currently it supports only limited FPGA architectures (mainly Xilinx Virtex 4, 5 and 6 architectures) and depends on ModelSim for simulation support [34].

[1]Source code and implementation details available at https://code.google.com/p/resim-simulating-partial-reconfiguration/wiki/ReSim

2.6 Limitations of Partial Reconfiguration

The process of incorporating partial reconfiguration in FPGA based designs increases the complexity of the design flow manifold. The limitations of using partial reconfiguration are enumerated in this section.

1. Current available commercial tool chains do not support reconfiguration of all component types. For instance global clocks and clock modifying circuitry must reside in the static region of the design [26].

2. The additional complexity on account of partial reconfiguration increases the time required by the design tools for synthesizing and implementing the design [1]. Due to the involved complications, current design tools do not support all the advances in the technology. E.g. current commercial tools do not support the grid and the slot style of placing the reconfigurable modules.

3. Both the design flows for partial reconfiguration — difference based and module based — have their own limitations ranging from routing challenges to restrictions in loading arbitrary design configurations. These have been described in brief in their individual sections - 2.2.1 and 2.2.2.

4. Use of partial reconfiguration introduces threat of security breaches by allowing unauthorized personnel to modify operating hardware designs. The process of mitigating the security risks is an ongoing research topic [35, 36].

2.7 Applications of Partial Reconfiguration

There are innumerable applications which use partial reconfiguration and examples where they would be very useful. This section briefly mentions a few of the applications.

Various applications using partial reconfiguration are presented in [1]. One of the applications involving use of partial reconfiguration in a network switch is briefly presented in this section. Network switch is a device with multiple ports used for interfacing different computer networks. Generally, a switch supports several communication protocols on each of its ports. For this purpose, each of the ports is connected to multiple interfaces each implementing a specific protocol. At run time, the interface for the required communication protocol is used. However this leads to unnecessary wastage of FPGA area as each port has to implement several communication interfaces, most of which will never be used. The FPGA resource consumption can be reduced by usage of partial reconfiguration as shown in Figure 2.7. In this implementation, each port of the network switch is connected to an empty reconfigurable container. The bitstreams corresponding to all the interface implementations are stored in some non volatile memory. At runtime, ports can use any of the communication interface by loading the appropriate bitstream. Such an implementation also provides additional flexibility in terms of extending the support to a new protocol. This would just require adding an additional bitstream to the non volatile memory.

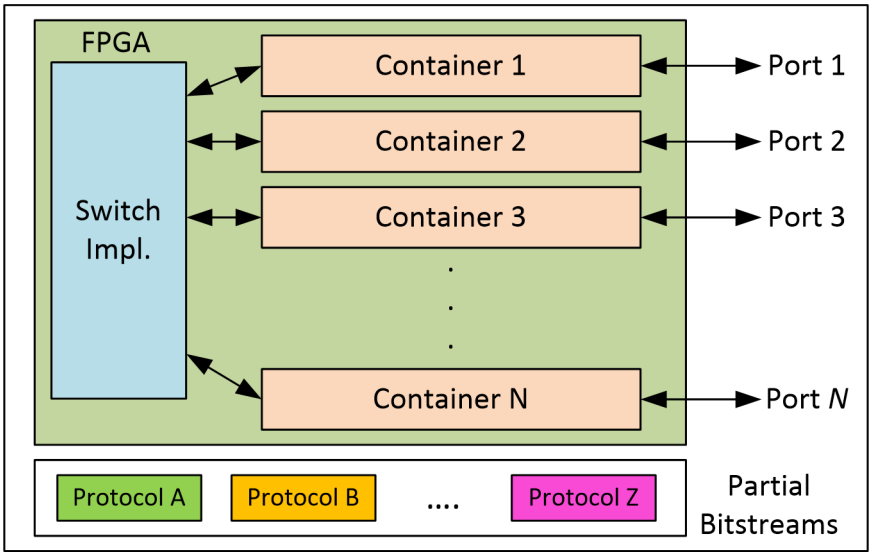


Figure 2.7: Network Switch - Implementation using Partial Reconfiguration

The partially reconfigurable network switch uses partial reconfiguration with rather coarse granularity. Partial reconfiguration has also been used with finer granularities for development of processors which use reconfigurable fabric for implementing specific instructions [6, 7]. Rotating Instruction Set Processing Platform (RISPP) is an example of one such platform. RISPP is a platform with a processor which uses a reconfigurable fabric for implementing hardware accelerators for expediting the execution of a few selected instructions. In contrast to all the other state-of-the-art reconfigurable architectures, RISPP takes runtime decisions on executing an instruction using the general datapath of the processor or using the accelerator implemented in the reconfigurable fabric. RISPP uses a runtime system which selects an implementation of an instruction from the available alternatives based on the monitored frequency of execution of the instruction and the size of the reconfigurable fabric in use.

Summary

Partial reconfiguration of an FPGA deals with modification of a design while it is operating on the FPGA. Hardware designs must be divided into static logic and reconfigurable logic for implementation with partial reconfiguration. Designs using partial reconfiguration benefit by reduction in power and resource consumption. Implementations using partial reconfiguration can be used for improving fault tolerance and for implementing applications based on artificial intelligence. There are two types of design flows for implementing applications using partial reconfiguration — Difference based flow and Module based flow. Difference based flow is used for making small modifications to the design while the module based flow is used for replacing hardware modules in the design. Tool chains supporting designs using partial reconfiguration are available from different vendors.

Chapter 3

Image Processing with Partial Reconfiguration

Contents

Image processing is ubiquitous these days with areas of applications ranging from the automotive domain to medical diagnosis. Today, most devices routinely employ image processing in some form or the other — state-of-the-art cameras rely on sophisticated image processing algorithms instead of photographer's skill, high-end laptops and smart phones use face recognition for security purposes to name a few. There is significant interest in the research community worldwide to improve the performance of devices utilising these complex image processing applications.

Image processing is a field of mathematics and computer science which deals with analysis of images to improve their quality or derive meaningful information from them. Formally, an image can be defined as a two dimensional arrangement of pixels, each of which has a value. The value defines the appearance of the pixel with respect to the model in use. The number of bits used to represent the value of a pixel defines its depth. The most popular model to represent pixels of coloured images is the RGB model, where each pixel value is composed of three spectral components, also known as channels - Red (R), Green (G) and Blue (B). These spectral components represent a cube, also known as the RGB cube, in the Cartesian coordinate system. Gray scale images can also be represented using this model, however in such cases, each pixel is composed of only one channel and their values represent the diagonal of the RGB cube. More details about the RGB colour model is available in pages 401 - 403

of [37]. Most coloured images being processed on computers have a 24-bit depth, with 8-bits representing each channel.

With the advent of sophisticated image sensors and High Definition (HD) video technologies, the number of pixels in an image has increased. The widely used High Definition Video format, 1080p, has dimensions of 1920 x 1080, i.e. 2,073,600 pixels, each pixel having a 24-bit depth. With such high data volumes, even simple image processing operations require a large number of computations. To improve the performance of these image processing algorithms, the spatial parallelism of the image data and the temporal parallelism of the algorithm has to be utilised by using multiple processor cores or customised architectures. This chapter discusses image and video processing algorithms which are interesting from the point of view of reconfigurable computing especially with respect to the use of partial reconfiguration.

3.1 Applications of Image Processing

The importance of image processing in today's world can be summed up with the fact, that there is almost no technical area that does not benefit from it. Image processing is used in medical applications for obtaining X-Ray images to diagnose the injuries to the skeletal system or for obtaining angiograms to locate clogged arteries and veins. It is used by disaster management departments for operational preparedness against storms, cyclones etc. by processing satellite images of the earth's atmosphere. Image processing applications are popular with law enforcement agencies world wide for applications ranging from forensic analysis to detecting counterfeit currency. Sky is the limit for applications of image processing. More applications of image processing are described in the Chapter 1 of [37].

3.2 Image Processing Algorithms with Partial Reconfiguration

This section describes image processing algorithms which are interesting for this thesis, from the point of view of implementation using partial reconfiguration. Applications which allow multiplexing of resources when implemented on FPGA are apt candidates for partial reconfiguration based designs. The algorithms covered in this section are similar to an image processing pipeline where each hardware module performs an image processing step and passes the result to the next module for the next step. During each image processing step, the FPGA resources responsible for performing the other steps are not used. This characteristic makes such algorithms suitable for implementation with partial reconfiguration.

3.2.1 Morphological Image Processing

Morphological Image Processing is the application of mathematical morphology on digital images for transforming them for various reasons like removing noise, isolating or joining regions, finding intensity bumps etc. All morphological image processing algorithms are based on two main operators, viz. Dilation and Erosion. The Dilation operator can be thought of

as *local maximum operator*, which causes bright regions within an image to grow. Erosion on the other hand can be thought of as *local minimum operator*, which results in reduction of bright regions within the image.

From the many morphological image processing algorithms, this thesis focusses on *Opening*, *Closing* and Morphological Gradient. The flow diagram for computing opened, closed and the morphological gradient of an image is shown in Figure 3.1 and 3.2.

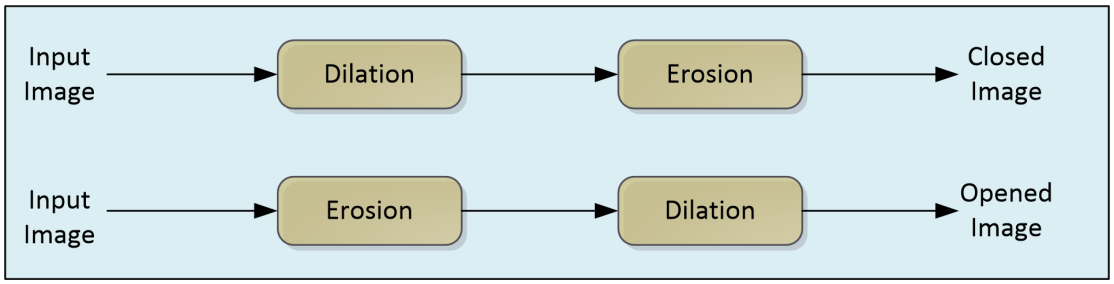


Figure 3.1: Flow diagram for Morphological Opening and Closing

- Opening : Opening is the application of the Erosion operator followed by the Dilation operator. It is mainly used for separating regions in binary images before counting them, e.g. counting of cells in a microscopic slide.
- Closing : Closing is the application of the Dilation operator followed by the Erosion operator. It is used as a step in other sophisticated image processing algorithms related to connected components. It is also used for reducing noise driven elements.
- Morphological Gradient : Morphological gradient is the absolute difference between the dilated image and the eroded image. This operation is used mainly for isolating perimeters of blobs.

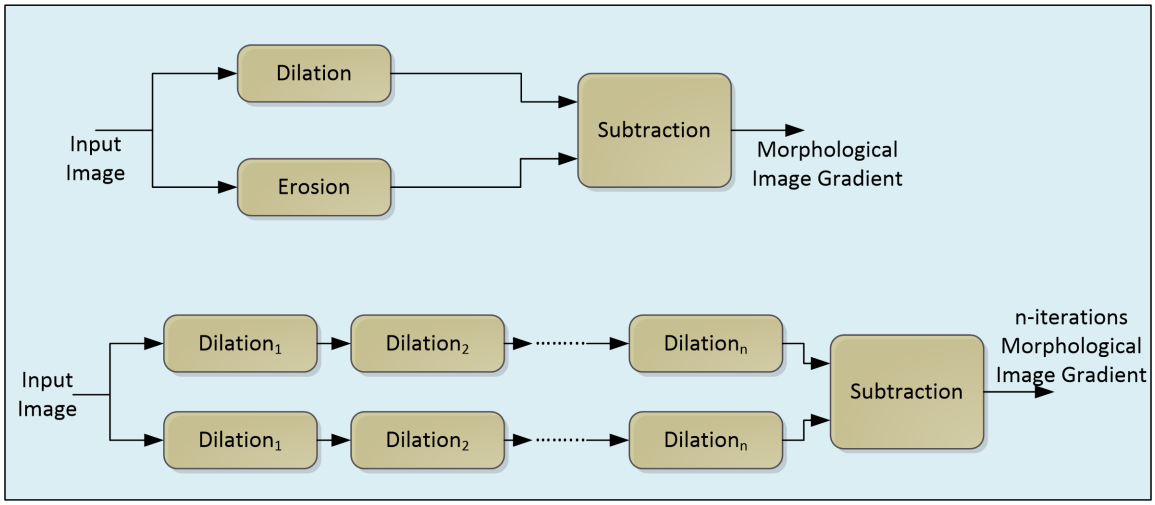


Figure 3.2: Flow diagram for Morphological Image Gradient

(a) Reference Image (b) Closing (c) Opening (d) Morph. Gradient

Figure 3.3: Morphological Image Processing - Opening and Closing

Figure 3.3 shows the effect of these operations on a standard test image[1]. The morphological image operations in the figure are executed on a 3 x 3 pixel neighbourhood. As seen, opening results in separation of regions, while closing removes noise segments by blurring. The morphological gradient operation isolates the perimeters of the bright regions.

By means of partial reconfiguration, morphological image operations can be implemented on an FPGA with two hardware modules - one performing Dilation and the other performing Erosion instead of a dedicated module performing the entire operation. The use of partial reconfiguration for the operations *Opening* and *Closing* provides certain specific advantages,

1. *Opening* and *Closing* differ only by the sequence in which the Dilation and Erosion operators are applied. Using partial reconfiguration for implementing the functionality with two hardware modules, as mentioned, makes it possible to have a flexible system which can achieve *Opening* as well as *Closing* without any changes in hardware design. With minor changes in scheduling of the cores i.e. changing the sequence in which cores are loaded on the FPGA, the functionality can be changed from *Opening* to *Closing* and vice-versa.

2. Opening and Closing differ from other operations when it comes to iterative processing. When *Opening* is to be performed iteratively twice, the implied sequence of operation is not Erode-Dilate-Erode-Dilate. Instead as shown in Figure 3.4, the required sequence of operations is Erode-Erode-Dilate-Dilate. With a partially reconfigurable system, the number of processing iterations can be changed without major changes to the hardware. In absence of partial reconfiguration, the hardware would require resynthesis. As shown in Figure 3.2, this advantage is also applicable for the morphological image gradient operation.

Mathematical details of the Dilation and Erosion operators along with all the other morphological image operations are given in the Chapter 5 of [17]. The implementation details of these applications using partial reconfiguration are covered in Chapter 5 of this report.

[1]Reference image - Lenna, Source - http://en.wikipedia.org/wiki/File:Lenna.png, License - Use of this picture is "overlooked" and by implication permitted by the copyright holder - Playboy

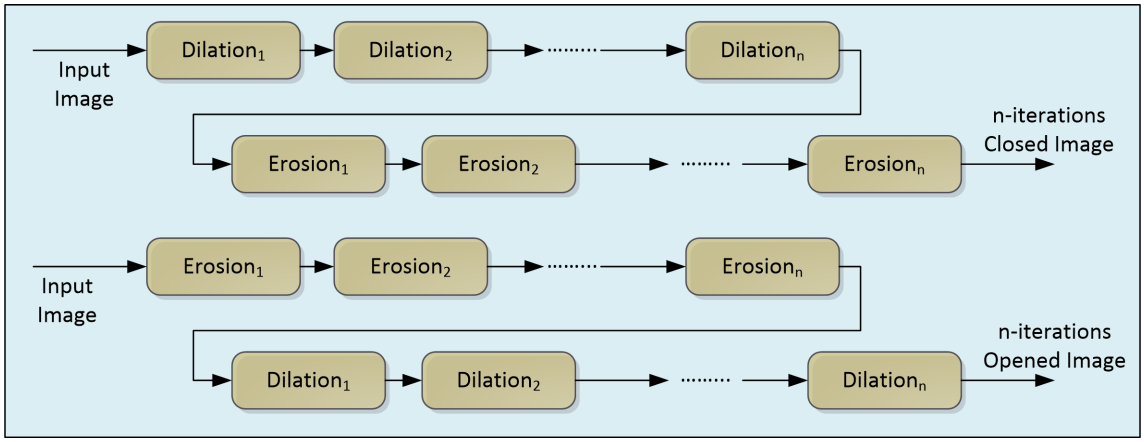


Figure 3.4: Flow Diagram for iterative Morphological Opening and Closing

3.2.2 Lane Detection

Lane detection is one of the more popular applications of image processing and is widely used in the Lane Departure Warning Systems, which warns drivers if they change lanes without proper indication [38]. Figure 1.1 shows the use of image processing for identifying lanes on a highway.

Many different algorithms have been proposed for lane detection [39, 38]. Figure 3.5 shows a simplified flow diagram of the steps involved in lane detection, as proposed in [39].

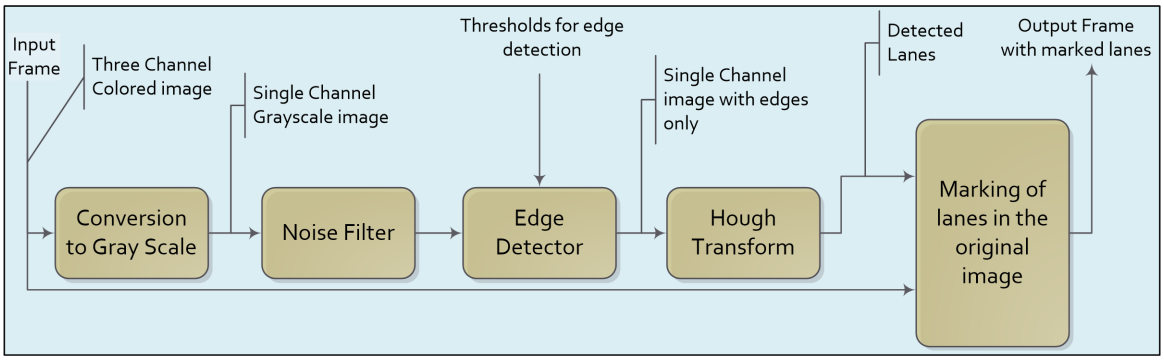


Figure 3.5: Steps involved in lane detection

For detecting lanes from an input image, it is first converted to gray scale and filtered for removing noise elements. Edges in the resulting image after noise removal are detected using edge detectors like Sobel or Canny. Finally, Hough transform is used to separate straight edges representing lanes from the others. Detected lanes can be marked on the input image for display. Thus, the algorithm forms an image processing pipeline with five processing steps. Details on each of the mentioned image operations are provided in [17]. Each step of the image processing pipeline can be performed using a separate hardware module which is loaded into the FPGA based on a schedule, thus implementing it using partial reconfiguration. The

implementation of the Lane Detection algorithm using partial reconfiguration is explained in Chapter 5 of this thesis.

3.2.3 Background Subtraction

Background Subtraction is an image processing step, mainly used in video security applications for separating the foreground objects from the background for analysis. The background is defined as the part of the frame which remains static or in a state of constant motion. From the point of view of a security camera at an Automated Teller Machine, the background is perhaps the ceiling of the room where the machine is placed and the customer using the machine is the subject, whereas for a camera at a traffic signal, the moving vehicles are considered as background. With background subtraction, the uninteresting parts of the frame are removed to focus the processing efforts at the actual subject.

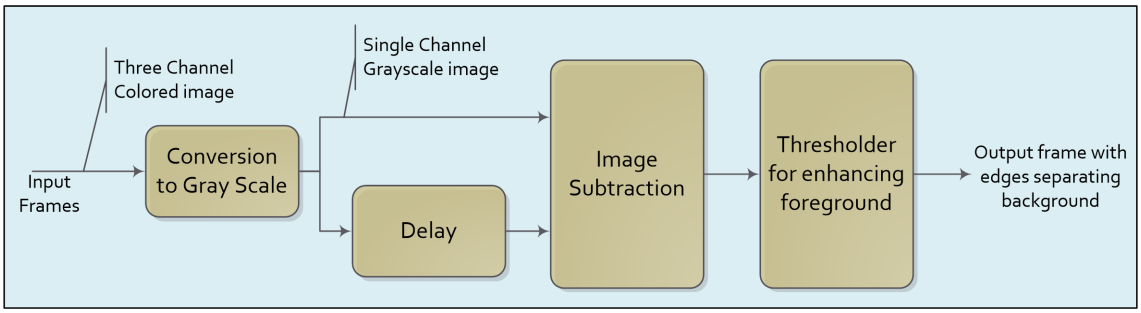


Figure 3.6: Steps involved in background detection by means of frame differencing

While there are many image processing algorithms for modelling the background of a scene and subtracting it from the frame, this thesis considers one of the simpler algorithms for implementation. Frame differencing is based on subtracting an image frame from a frame occurring after some delay and then enhancing the difference between them. Anything that is static between the two frames will not be visible in the final result. The resulting image would contain edges of the moving objects and can be used for indicating regions of motion. The flow diagram depicting the steps involved in frame differencing is shown in Figure 3.6. In short, frame differencing can be implemented by converting the input image to a gray scale image and subtracting it from the gray scale image of the previous frame. The resulting difference may be thresholded for enhancing the regions of motion.

3.3 Image Processing in Software

This thesis is interested in software image processing libraries which allow the design of image processing applications to later convert these to hardware descriptions. MATLAB and OpenCV suit the purpose well. MATLAB provides models which can be synthesized into hardware descriptions while OpenCV is implemented in C/C++ which can be synthesized by high level synthesis tools. This thesis utilises OpenCV to generate hardware descriptions

for image processing applications because MATLAB is proprietary and requires additional licensing.

3.3.1 Open Computer Vision (OpenCV)

Open Computer Vision, popularly known as OpenCV[2], is a widely used computer vision library for image and video processing. Its origin lies in an Intel research initiative for making computer vision infrastructure universally available. One of the primary goals of the OpenCV development was to provide open and optimised code for basic vision infrastructure. It is highly optimised for performance compared to its peers. The performance comparison of OpenCV against the other available open source vision libraries is given in Chapter 1 of [17].

OpenCV not only provides primitives for image processing, but also provides a platform independent graphics tool kit known as HighGUI. HighGUI abstracts the operating system calls for accessing devices like cameras to query frames, for accessing file systems to load or save images and for accessing the windowing system to display images and videos. OpenCV provides high level image processing operators for operations like smoothing, segmenting, tracking etc. Detailed documentation for the API's provided by OpenCV is available on the web and in [17].

3.4 Image Processing with FPGAs

Image processing has been a potent driver effecting improvements in the reconfigurable architectures mainly because of the unacceptable performance of the general purpose computing. The introduction of this chapter already explains the inability of general purpose processor architectures to meet real time processing constraints of image processing algorithms like the frame rate due to the large amount of data which needs processing along with high bandwidth requirements with the memory where the images are stored. On the other hand specialised processor architectures can exploit the inherent parallelism in the image processing algorithms and provide better performance. In accordance to Amdahl's law, which states that the speed up of a system is only limited by its sequential part, high speed-ups can be achieved for image processing algorithms, most of which can be highly parallelized using reconfigurable architectures.

Image processing algorithms are classified in Chapter 6 of [18] as :

1. Local Algorithms - These work on data sizes smaller as compared to the size of the image and are local temporally and spatially, e.g. Convolution, Thresholding etc. Generally these algorithms can be optimised to work with a single pass through the image data. All the algorithms described in Section 3.2 belong to this category.

[2]http://opencv.org/documentation.html

2. Global Algorithms - These depend on the data of the entire image and require multiple passes through the data. E.g. Fast Fourier Transform, statistical histogram techniques etc.

The key to achieve high speed ups with specialised architectures is to reduce the number of times the data is accessed. Hence higher speed ups can be obtained using customised hardware for implementing local algorithms. Detailed information for implementing local image processing algorithms like Convolution, Morphological Operations etc. on FPGAs is given in Chapter 6 of [18]. Such architectures are generally designed at Register Transfer Level (RTL) and hence consume lot of time and effort for development and verification. Because of the higher abstraction level provided by high level synthesis for design, it is a faster method for the development of such architectures.

As mentioned in Chapter 1, this thesis does not deal with the development of customised hardware for image processing applications, rather it seeks to explore the usage of partial reconfiguration in implementing them. Hence this thesis prefers using Intellectual Property (IP) cores which are pre-designed and verified or which can be easily synthesized from a high level description instead of developing it from scratch.

3.4.1 High Level Synthesis Video Library

In order to simplify the development of FPGA based hardware architectures for image processing, Xilinx Inc. released a C++ library of synthesizable functions. This library of functions, known as the Vivado High Level Synthesis (HLS) Video Library, contains data structures and functions corresponding to the ones implemented in OpenCV. This library was released with the v2013.1 of Vivado High Level Synthesis tool. C++ programs invoking these functions can be converted to IP cores performing the same functionality using high level synthesis, as described in Section 4.2.1. Currently in its second release, the functionality of HLS Video Library is limited compared to OpenCV but still provides enough functions for developing IP cores which function as image processing pipelines.

Along with functions for processing images in FPGAs, the library also provides synthesizable functions to convert an image into sequence of raw data, stream, and vice-versa to help interface the generated IP core with other cores. Since Xilinx Inc. has adopted Advanced eXtensible Interface (AXI) as standard interconnection for its recent products, these functions convert images to/from AXI4-Streams[3], to be specific. These functions abstract the programmer from issues pertaining to interfacing the cores and setting up communication between them.

The HLS Video Library is important from the point of view of the thesis as it provides a mechanism to generate of image processing cores, which are required for implementing image processing applications with partial reconfiguration. Further, the fact that most functions from HLS Video Library correspond to functions in OpenCV, can be exploited for easier validation of the generated IP cores [40]. All the details on utilising the Vivado High Level

[3]Details on AXI and AXI4-Streams are provided in Chapter 4

Synthesis tool to create the image processing IP cores are described in Chapter 4. A detailed description of all IP cores generated using HLS Video Library along with their validation is provided in Chapter 5.

Summary

Image Processing applications are widely used in today's world, however given the computational complexity of these algorithms, they are not suitable for execution on general purpose processors. Usage of customised hardware architectures for image processing is very popular and has been a key driver in development of these architectures. Image processing algorithms are implemented in many open source libraries which can be utilised for creation of customised hardware by means of high level synthesis. Image processing applications like Morphological Image Processing, Lane detection and Background Subtraction are suitable for implementation using partial reconfiguration as a part of the thesis. These algorithms can be implemented with the help of the functions provided by OpenCV, one of the most widely used image processing libraries. The HLS Video Library - a synthesizable version of Open CV, developed by Xilinx Inc., can be utilised for synthesising IP cores of for image processing applications from software implementations.

Chapter 4

Zynq-7000 for Partial Reconfiguration

Contents

One of the primary goals of this thesis is to evaluate the effectiveness of partial reconfiguration as a design pattern on runtime reconfigurable architectures with a processor core. This chapter gives a brief introduction of reconfigurable architectures which embed a processor and introduces Zynq-7000, a recent product from Xilinx Inc., which is an example of the mentioned architecture.

Embedded processors in reconfigurable architectures can be classified as soft processors, which are built using FPGA resources, and as hard processors, which are fabricated on dedicated silicon. Such architectures allow optimisation of tasks between hardware and software to maximise performance and efficiency. Software bottlenecks can be offloaded to hardware accelerators which are usually connected to the embedded processor with low latency channels [41]. Soft processor cores cost in terms of FPGA logic resources only and hence can be instantiated on any FPGA with sufficient resources. Hard processors are available on certain product families only, e.g. Arria V SoC and Cyclone V SoC families from Altera Corp. and Zynq-7000 family from Xilinx Inc. Hard processors provide better CPU performance compared to the soft processors and hence preferred for the thesis.

Zynq-7000 family is chosen for the implementation of the image processing applications introduced in Chapter 3 considering the availability of prototype boards and Xilinx tool chain

licenses within the OTERA[1] project. The Zynq-7000 family targets high end embedded systems with a processor centric approach for reconfigurable computing. Zynq-7000 supports wide application domains ranging from automotive applications to industrial automation. The upcoming sections enumerate the features of Zynq-7000 family which are attractive from the point of view of the thesis and further explains the tool chain required for implementing partial reconfiguration based image processing applications on Zynq-7000 devices.

4.1 Zynq-7000 All Programmable SoC - Salient Features

The Zynq-7000 family consists of devices which integrate a dual core ARM Cortex A9 with a reconfigurable fabric based on 28 nm technology. This combination provides high I/O bandwidth with low latencies that cannot be matched by the older two chip solutions with the processor and the reconfigurable fabric on different devices.

The brain of the device is the Application Processing Unit (APU) containing two ARM Cortex A9 processors along with their NEON co-processors for supporting media and signal processing with its Single Instruction, Multiple Data (SIMD) architecture. The ARM processors are high performance and low power cores which contain separate 32 KB L1 caches for data and instructions and a shared 512 KB L2 cache along with 256KB on-chip memory. Hardware support for maintaining coherency between the multi-level caches is available in the device. The APU, along with other peripherals for I/O, forms the Processing System of the device.

The reconfigurable fabric on these devices is fabricated using the state-of-the-art high-performance low-power (HPLP), 28 nm, and high-k metal gate (HKMG) technology. The reconfigurable fabric provides a wide range of user configurable resources like the Configurable Logic Blocks (CLBs), 36 Kb Block RAMs, Digital Signal Processors with a resolution of 48-bits, Analog to Digital Converters, Configurable I/O blocks etc. All these reconfigurable resources within the device form the Programmable Logic.

The Processing System and Programmable Logic communicate with each other through high bandwidth interconnects based on ARM Advanced Microcontroller Bus Architectures (AMBA). AMBA is a bus architecture introduced by ARM Ltd. which specifies on-chip buses used in System on Chip architectures. The most important of the buses defined by the latest version of AMBA, AMBA v4, with respect to Zynq-7000 family is the Advanced eXtensible Interface (AXI). As shown in figure 4.1, there are four high performance AXI Master ports in the Programmable Logic, four general purpose AXI ports (two masters and two slaves) and an Accelerator Coherency Port (ACP) based on AXI. Together, all these interfaces provide a high bandwidth for communication between Processing System, Programmable Logic and external memory. Figure 4.1 shows the main components (Processing System and Programmable Logic) of the Zynq-7000 device. Further notes on the features of the Zynq-7000 platform are given in [42].

The Processing System, which is encapsulated by AMBA interconnects, can be easily interfaced with Xilinx Intellectual Property (IP) cores, implemented in Programmable Logic, as

[1] Refer Chapter 1 for details

Xilinx has adopted AXI as the standard interface for its IP cores. AXI4, the latest version of AXI, as specified by AMBA v4, consists of three interfaces,

1. AXI4 : For high throughput communication including burst mode transfers with memory mapped peripherals,

2. AXI4-Lite : For low throughput and simplified communication, like reading and writing of status/control registers etc. with memory mapped peripherals,

3. AXI4-Stream : For high speed streaming data between two IP cores.

IP cores with AXI4 interfaces, from different sources, can be integrated in the Programmable Logic and interfaced with the Processing System. Based on the bandwidth requirements, suitable AXI4 interfaces can be used for this interfacing [43]. Details on programming the ARM cores and interfacing them with AXI4 compliant IP cores implemented in the Programmable Logic are provided in [30].

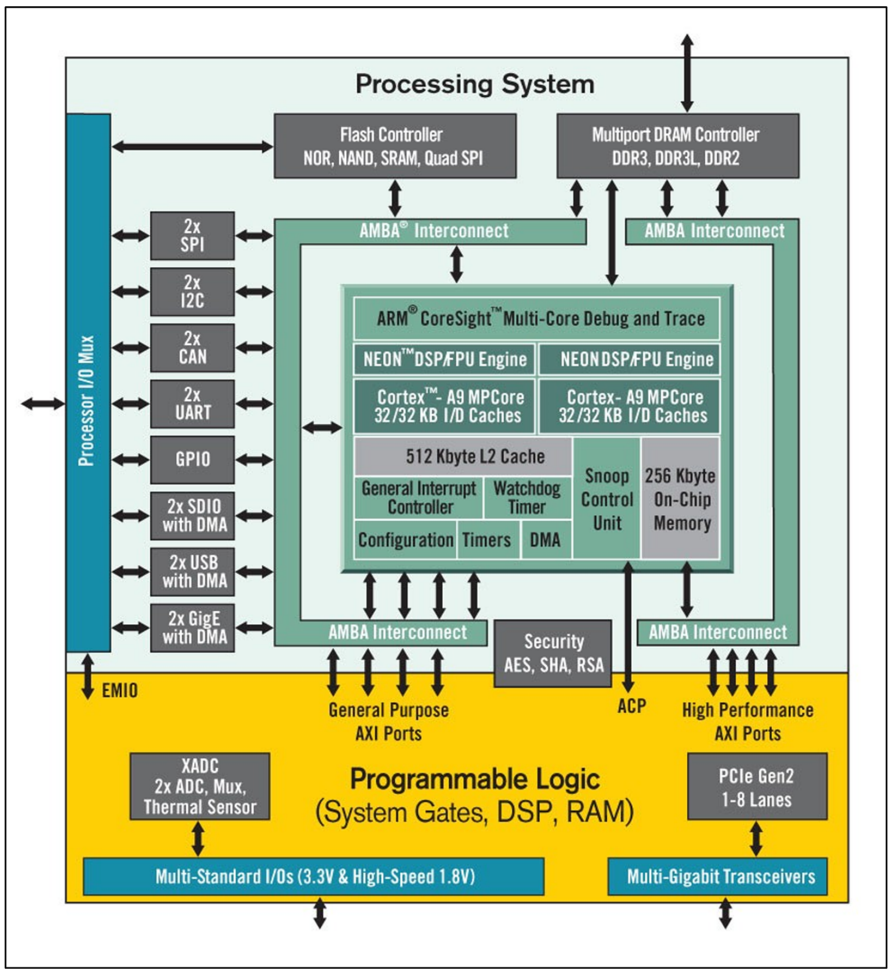


Figure 4.1: Zynq-7000 Extensible Processing Platform [44]

The most attractive feature of Zynq-7000 from the point of view of this thesis is the presence of a Processor Configuration Access Port (PCAP) which allows full as well as partial reconfiguration of the Programmable Logic by the Processing System. This enables configuring required hardware accelerators into the Programmable Logic at runtime by means of software scheduler executed on the Processing System. Zynq-7000 platform is also supported by a huge eco-system of operating systems including several Linux distributions. Operating System support eases the implementation effort of complex systems, like the ones implemented as a part of this thesis, by providing adequate levels of abstraction to interact with the accelerators implemented in the reconfigurable fabric. The success of this platform in implementing complex image processing algorithms like Feature Detection has already been proven by [45].

4.1.1 Zedboard

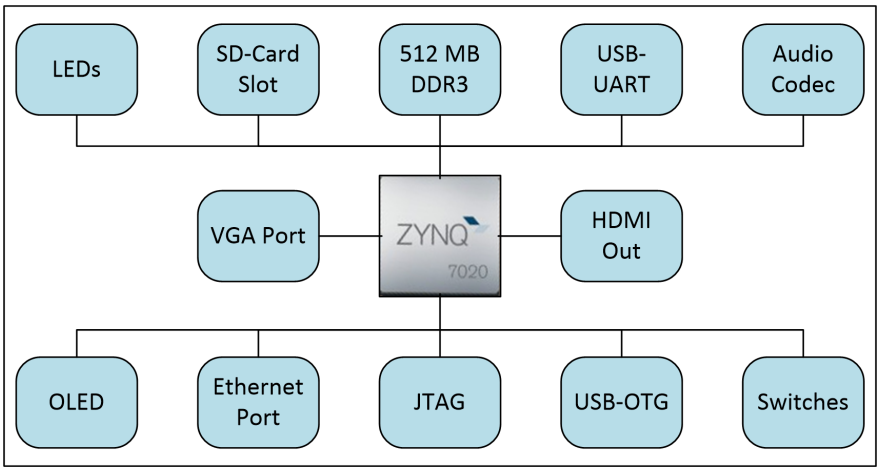


Figure 4.2: Zedboard : Block diagram with important peripherals

Zedboard is a low cost development board which uses a device from the Zynq-7000 family. It uses the device Z-7020 (part number - XC7Z020-CLG484 with speed grade -1) and has all required peripherals for implementation of image processing applications. This device contains reconfigurable fabric from the Artix-7 family which provides highest system performance per dollar per watt among the others in the 7-series of Xilinx programmable logic and shows 50% power reduction compared to the previous generations [23]. The Zedboard comes with 512 MB of onboard DDR3 memory and a SD-Card slot capable of hosting Linux file systems [46]. Figure 4.2 shows a block diagram depicting the important peripherals on the board.

4.2 Tool Flow

This section gives a short description of the tool flow involved in designing an embedded processor based system which utilises partial reconfiguration. The choice of Zedboard dictates the usage of Xilinx tool chain to avoid any incompatibilities. Xilinx Integrated Software

Environment (ISE) Design Suite provides all the features required for the creation of highly efficient core based designs and system integration along with support for partial reconfiguration and debugging. All the design and implementation involved in the thesis utilised the latest available version of the Xilinx ISE Design Suite - v14.6. Figure 4.3 displays the tool flow for generation of bitstreams implementing image processing applications using partial reconfiguration.

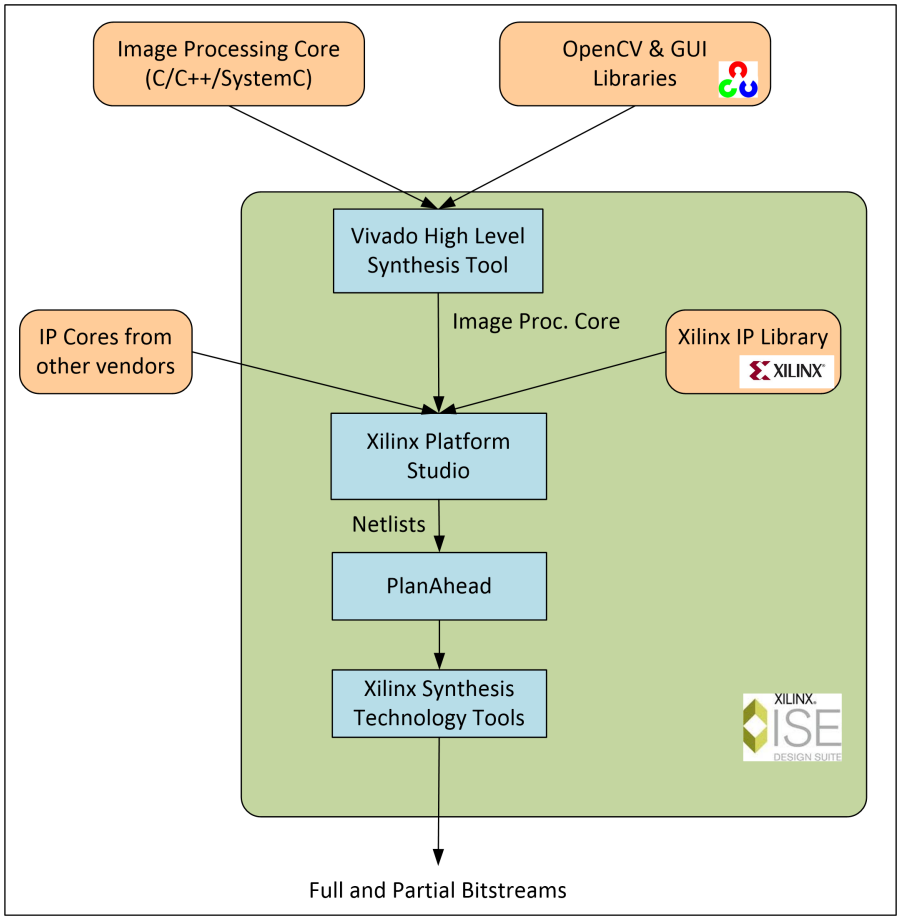


Figure 4.3: Xilinx Tool Flow

4.2.1 Vivado High Level Synthesis

Xilinx ISE Design Tool Suite provides Vivado High Level Synthesis (HLS) tool for creation of functional specifications for an FPGA based design at an abstraction level higher than the Register Transfer Level (RTL) using C, C++ or SystemC. Vivado HLS tool is capable of converting the functional specifications to their RTL equivalent (Verilog or VHDL), which can then be synthesized for any target FPGA architectures. The tool also provides features for pre-synthesis and post-synthesis validation of designs by means of test benches written in C or C++. Thus, Vivado HLS tool allows the users to focus on specifying the functionality of

the system instead of its logic implementation enabling faster design and validation of systems [47].

Vivado HLS can utilise the HLS Video Library, described in Section 3.4.1, for synthesis of image processing functionalities of OpenCV into FPGA based designs and their validation using the latest OpenCV library. Vivado HLS provides the feature of wrapping the generated RTL descriptions with standard interfaces like FIFO, AMBA etc. and exporting it as an Intellectual Property (IP) core to other tools in Xilinx ISE Tool Suite.

This thesis utilises Vivado HLS for generating IP cores wrapped with standard interfaces for performing image operations. These generated IP cores are exported to Xilinx Embedded Development Kit (EDK) after validation.

4.2.2 Xilinx Embedded Development Kit (EDK)

Xilinx EDK is a tool available within the Xilinx ISE Tool Suite used for design of embedded processor based systems. The EDK consists of three components

- Xilinx Platform Studio (XPS) : Used for the design of the embedded processor hardware.
- Xilinx Software Development Kit (SDK) : Used for development of drivers and application softwares to be executed on the embedded processor.
- Xilinx Intellectual Property Library : Provides verified IP cores which can be used as building blocks for the embedded processor hardware.

Embedded processor systems incorporate one or more processors with many other peripherals and memory blocks which are interconnected with processor buses. Software programs for different applications, residing in the memory, can be executed on this customised hardware. The EDK provides a development environment for the complete system - for customising the hardware as well as for developing the application softwares. The XPS is used for integrating and configuring different IP cores (processor as well as peripherals), available from the Xilinx IP library and other sources. The designed hardware can then be exported to the SDK which provides basic drivers for the application software, thus simplifying the process of application software development [48].

This thesis uses EDK only for the design of the embedded hardware for implementing image processing applications. Using XPS, the Processing System is interfaced with all the necessary IP cores executing image processing applications. The thesis utilises the IP cores available from the Xilinx IP Library for purposes ranging from data transfer to debugging. After all the IP cores are interfaced, XPS is used for generating the netlists of the designed hardware for the partial reconfiguration based design flow using the PlanAhead tool.

The detailed hardware design, as assembled using XPS, is explained in Section 5.1. The software being executed by the Processing System, responsible for the reconfiguration and implementation of image processing applications, is explained in Section 5.2.

4.2.3 PlanAhead

PlanAhead is a design and analysis tool in the Xilinx ISE Design Tool Suite which supports module-based partial reconfiguration. PlanAhead features post-synthesis tools which processes synthesized netlists and generates partial and full bitstreams for designs based on partial reconfiguration. PlanAhead implements module based partial reconfiguration by allowing the users to designate areas of the FPGA fabric as reconfigurable partitions. Parts of the input netlists can be designated as reconfigurable modules and assigned to these reconfigurable partitions. In-built design rule checker ensures the matching of the interfaces of the reconfigurable partitions and the reconfigurable modules assigned to it. Bitstreams are generated by means of design configurations which map reconfigurable modules to the partitions. Current version of the tool allows assigning only one reconfigurable module per reconfigurable partition, hence only island style of placing the partitions is possible [1]. Multi-island style of placing the partitions is achieved by creation of unique partial bitstreams for the reconfigurable module, each targeting a partition where module is to be loaded. Thus if an IP core is designed to be loadable on N partitions, N partial bitstreams must be generated with each bitstream targeting one partition.

PlanAhead is utilised by this thesis to create reconfigurable containers in the hardware assembled using XPS. The IP cores, generated using the Vivado HLS tool, responsible for performing image operations are marked as reconfigurable modules. PlanAhead generates partial bitstreams for all these IP cores for implementing the image processing applications.

4.3 Design Choices

As mentioned in Section 4.1, Zynq-7000 devices contain a Processor Configuration Access Port (PCAP) which can download partial bitstreams into the reconfigurable fabric. It is, hence, not necessary to integrate hardware components like Internal Configuration Access Port (ICAP) in the design to do the same. Also, encapsulation of Processing System with Advanced Microcontroller Bus Architectures (AMBA) Interconnects makes it easier to interface it with any IP core having AMBA interfaces. Finally, the tool chain from Xilinx Inc., described in Section 4.2, makes available all the tools required for designing a complex image processing system using partial reconfiguration. Thus, the choice of Zynq-7000 device for implementation simplifies the design considerations.

The main design considerations for the implementation are:

- Achieve high bandwidth for communication between the Processing System and the Programmable Logic,
- Perform module based partial reconfiguration, considering the modules in the reconfigurable logic are complex and differ significantly,
- Use island style of placing the reconfigurable containers as the current version of PlanAhead cannot support other styles,

- Use software controlled partial reconfiguration of the Programmable Logic via the PCAP,
- Use IP cores generated from the high level synthesis of image processing operations using the HLS Video Library, described in Section 3.4.1.

Of all the design goals, the goal to achieve high bandwidth between Processing System and the Programmable Logic is the most critical considering its impact on performance and design approach. The two possible design options, explored during the design phase of the thesis, are explained in the following sections.

4.3.1 Design with FIFOs - Xillinux

Xillinux is a proprietary Linux distribution from Xillybus Ltd. which extends the Zedboard into a Linux system based on Ubuntu 12.04 LTS. Unlike the Linux distributions for desktops, Xillinux is a combination of software and hardware, software executed by Processing System and hardware implemented in Programmable Logic. The hardware design features a VGA adapter which can be connected to any compatible monitor for display. Further, the USB-OTG port of the Zedboard can be attached to any Linux compliant keyboard and mouse which it turns into a full fledged graphical desktop. The selling point of Xillinux is the presence of an IP core, Xillybus, in its hardware design, which provides FIFO interfaces between Processing System and Programmable Logic. Xillinux software comes with the required Linux drivers which provides the programs running in Linux userspace interfaces for interaction with the FIFOs. Data communication between Processing System and Programmable Logic, thus simplifies to just reading from and writing to certain device files in the Linux file system. Figure 4.4 shows the FIFO based architecture implemented in Xillinux. Xillybus Ltd. also provides a web interface for customising the FIFO interfaces with respect to the data width and bandwidth expectations. Customised cores can be downloaded from the web interface of Xillybus Ltd. almost immediately[2]. The standard FIFOs available with Xillinux have a width of 32-bit and provide a bandwidth of about 200 MB/sec.

As the goals of the thesis are limited to evaluation of partial reconfiguration on FPGAs, the usage of Xillinux and its proprietary IP cores for implementation should not be any cause of licensing based concerns. Xillinux is free for any use, as long as the usage reasonably matches the term "evaluation". There are no plans to extend the implementation for any purpose beyond evaluation and hence free evaluation license would suffice from the point of view of this thesis.

By providing ready to use communication channels between the Processing System and Programmable Logic, Xillinux simplifies the application development on the Zedboard. The application logic on the other side of the FIFOs could be IP cores responsible for image processing. Detailed tutorials for synthesising cores with FIFO interfaces are available in Xillinux documentation. Images to be processed can be sent to the image processing cores through the Write FIFO and the processed images can be read back using the Read FIFO. These image processing IP cores can be loaded on to the reconfigurable fabric based on requirement

[2]URL for the web interface : http://www.xillybus.com/custom-ip-factory

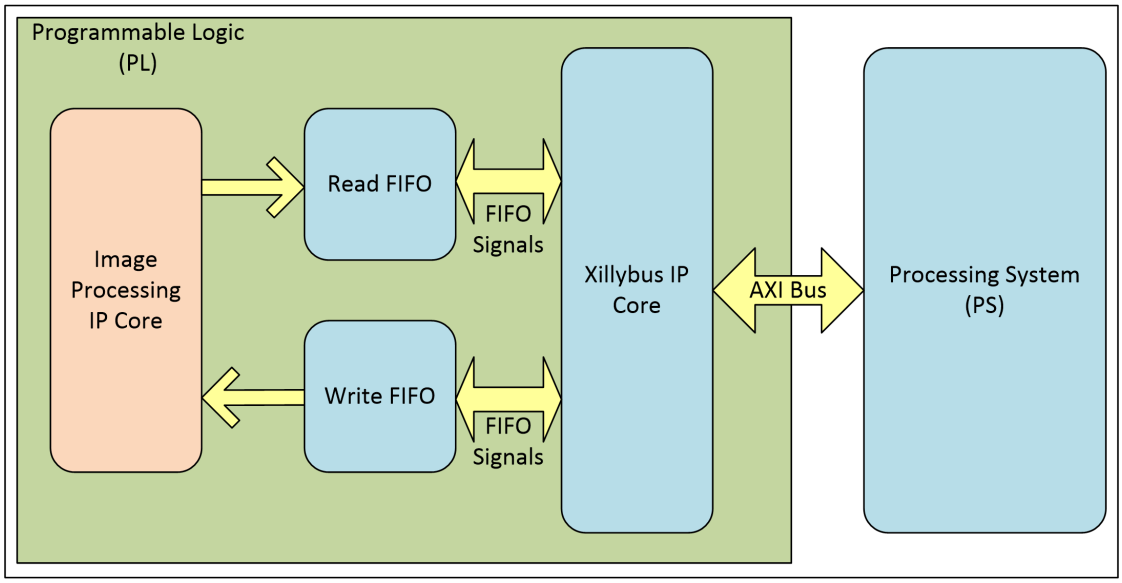


Figure 4.4: Xillinux FIFO based architecture on Zedboard [49]

using runtime partial reconfiguration, thus forming the reconfigurable part of the design. The Xillybus cores along with the FIFOs and the VGA cores would form the static logic of the design.

Using the FIFO based architecture provided by Xillinux though have some drawbacks also. Firstly the Xillybus core, which manages all the FIFOs to and from Programmable Logic, is available only as a netlist. The source code and the behavioral models of the core, which may be required for simulation during development, are not available from the vendor, Xillybus Ltd. The main drawback of this design lies in the fact that processor utilisation will be very high as the Processing System has to consistently write to and read from FIFOs, just like in Programmed I/O. For the thesis implementation, the Processing System has to read the raw data of the huge images kept in the external memory and write it to FIFO and vice versa through one of the high performance AXI port between the Processing System and Programmable Logic. This frequent access of the external memory by the Processing System results in high processor utilisation and high latency. A DMA based system would relieve the Processing System of its task of accessing the external memory and hence improve the performance of the implementation.

4.3.2 Design with AXI4-Streams - Video DMA

Xilinx Inc. has presented, through reference designs, the use of Video Direct Memory Access (VDMA) for implementing image/video based systems on the Zynq-7000 devices [11, 40]. Video DMA is a soft core which provides high bandwidth access between external memory and IP cores with AXI4-Streams. The Video DMA IP core has two AXI4-Streams, one for transferring data from memory to the IP core and other in the reverse direction, to interface

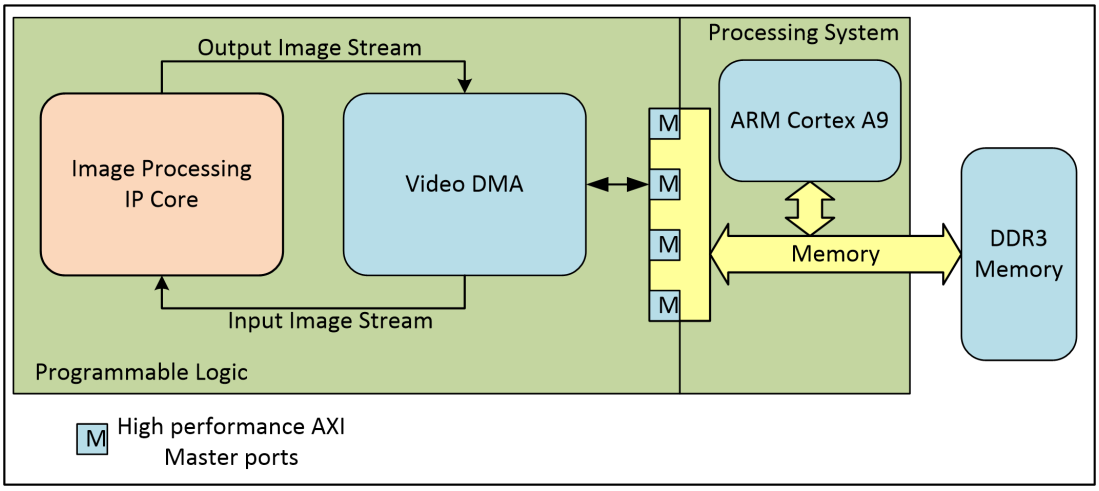


Figure 4.5: Using Video DMA for interfacing Processing System with application cores on Zynq-7000

with the IP cores. In the mentioned reference designs, Video DMA is configured to access image data kept in the external memory through one of the High Performance AXI ports in the Programmable Logic and stream it to the IP cores. These IP cores, designed for image processing, stream back the processed images to the Video DMA which writes it to the specified location in the external memory through the same AXI port. The figure 4.5 shows the use of Video DMA to interface image processing IP cores with the Processing System.

Use of Video DMA, which uses one of the high performance AXI port, provides a throughput of around 1,200 MB/s per channel compared to around 200 MB/s in FIFO based Programmable I/O described in Section 4.3.1 [30, 49]. In addition to the performance gain, the load on the Processing System reduces on using this approach. These reasons build a strong case in favour of using this approach as against the use of pre-designed FIFOs from Xillinux.

Summary

For evaluating the efficacy of partial reconfiguration based design, this thesis implements image processing applications on the Zedboard, which uses a Zynq-7000 device. Zynq-7000 has a dual core ARM Cortex A9, known as Processing System, and state-of-the-art reconfigurable fabric, known as Programmable Logic, which are interfaced with each other using high bandwidth AMBA interconnects. The image processing applications are implemented using the module based partial reconfiguration design paradigm in which the reconfigurable containers follow island style of placement. The process of reconfiguration is controlled through software executing on the Processing System. The primary challenge for designs using Zynq-7000 platform is achieving high bandwidth of communication between Processing System and Programmable Logic. FIFO based options provided by Xillinux, a Linux distribution, and Video DMA based design are two options explored for this purpose.

Chapter 5

Image Processing on Zynq-7000 with Partial Reconfiguration

Contents

Chapter 4 introduced Zynq-7000 as the selected device for implementing image processing applications using partial reconfiguration. This chapter presents the hardware and software design for the implementation of the image processing applications which were presented in Section 3.2.

Section 4.3 enumerates the design goals for the implementation along with possible approaches to achieve them. Of the two options, first using pre-designed FIFOs from Xillinux and second using Video DMA Intellectual Property (IP) core, this thesis uses the latter option considering the tremendous performance gain that is achievable as explained in Section 4.3.2. This decision also means that the optional goal of using DMA based solutions for implementation, as mentioned in Section 1.2.3, can be achieved. The hardware design for the implementation extends the Xillinux hardware, explained in Section 4.3.1, in order to utilise its inbuilt VGA functionality which is not provided by the other Linux distributions for Zedboard. The application software for the image processing applications is developed using the native GCC compiler of Xillinux, purely for convenience reasons. Thus, the thesis utilises Xillinux only as an operating system and not for its pre-designed FIFOs for data transfer between Processing System and Programmable Logic of the Zynq-7000 device.

5.1 Hardware Design

This section presents the hardware design for implementing the image processing applications on Zedboard using partial reconfiguration. IP cores performing image processing operations form the reconfigurable modules of the design. The static logic consists of Video DMAs for fetching images from the memory and reset circuitry for the image processing IP cores.

5.1.1 Image Processing Cores - Reconfigurable Modules

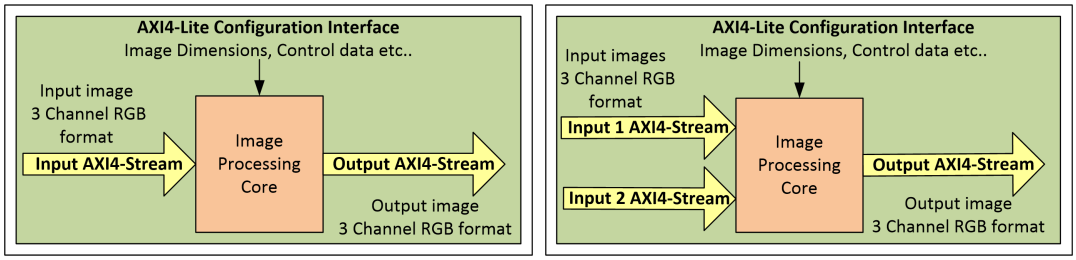
In order to implement image processing applications on Zedboard using partial reconfiguration, this thesis requires IP cores performing simple image operations. These IP cores form the reconfigurable modules of the implementation. Image processing applications can be implemented by loading the IP cores into reconfigurable containers based on an appropriate schedule. For e.g. referring to the morphological image processing, described in Section 3.2, *Opening* can be achieved by loading IP core capable of performing Erosion followed by IP core responsible for Dilation. The IP cores performing different image processing operations can be generated using Vivado High Level Synthesis tool with the HLS Video Library. Most cores generated for use in thesis have three bus interfaces for connecting with the other IP cores :

- Input Stream - An AXI4-Stream slave interface for acquiring images in 3-Channel RGB format[1]. Can be connected to the AXI4-Stream master interface of a Video DMA or another image processing core.
- Output Stream - An AXI4-Stream master interface for returning processed images in 3-Channel RGB format. Can be connected to the AXI4-Stream slave interface of a Video DMA or another image processing core.
- Configuration Interface - An AXI4-Lite slave interface for configuration. Configuration data includes the dimensions of the images along with control commands for starting or stopping the execution of the IP cores. The AXI4 master port to which this interface is connected gets the responsibility of configuring the IP core.

We name such cores with three bus interfaces as *TypeA* cores. However, few image processing operations require two input images for execution. Such cores are generated with two input stream interfaces, one for each image i.e. four bus interfaces in total. We name these cores as *TypeB* cores. The structure of these IP cores is shown in Figure 5.1.

The functionality of the generated IP cores is validated using the framework provided by the Vivado High Level Synthesis tool. For this purpose, test benches written in C++ are used. The Vivado High Level Synthesis tool uses this test bench for pre-synthesis and post-synthesis validations. The test benches use OpenCV primitives for the purpose of generating the reference images for validation. These images are compared with the images resulting from the execution of the IP cores under test. For IP cores not using floating point operations,

[1]Refer Chapter 3 for description on pixel value representations.



(a) TypeA - Core with three bus interfaces (b) TypeB - Core with four bus interfaces

Figure 5.1: Structure of Image Processing Cores generated using HLS Video Library

the validation is considered successful when the reference image generated by the OpenCV functions match the image generated by the IP core completely. For IP cores using floating point operations, the validation is considered successful, if the Peak Signal to Noise Ratio (PSNR) of the image generated by the IP core when compared to the reference image is not less than 30dB. A brief description of the PSNR algorithm for evaluating the quality of images is given in [50].

The description of all the generated IP cores are listed below. A brief description on using Vivado HLS tool for generating these image processing IP cores have been provided in Appendix A. The resource usage and performance for all these cores are provided in Appendix A.2.

Morphological Image Processing

The morphological image operations for which IP cores were generated are mentioned below.

Dilation - Local maximum operator on a 3 x 3 pixel neighbourhood.

Erosion - Local minimum operator on a 3 x 3 pixel neighbourhood.

Opening - Erosion followed by Dilation operation on the input image. Both operations performed on a 3 x 3 neighbourhood.

Closing - Dilation followed by Erosion operation on the input image. Both operations performed on a 3 x 3 neighbourhood.

Morphological Gradient - Absolute difference of the result of Erosion and Dilation operations on the input image. Both operations performed on a 3 x 3 neighbourhood.

All these image operations are elaborately explained in Section 3.2. All these operations require only one input image and are hence designed as *TypeA* IP cores.

Image Filters

These IP cores are designed to perform image filtering operations on a single image and are hence designed as *TypeA* cores. The image filters implemented are

Gaussian Filter This filter is mainly used for reducing the noise elements in an image. This IP core outputs the result of mathematical convolution of the image with a 3 x 3 Gaussian kernel. The details of the operation are provided in the Chapter 5 of [17].

Sobel Filter This filter is used for computing the derivative of the image and can be utilised for detecting edges. The generated IP core works on a 3 x 3 neighbourhood computing 1st order derivative along the X-axis of the image. The mathematical description of the Sobel filter is given in the Chapter 6 of [17].

General Image Processing Operations

These IP cores perform general image processing operations on the input image(s). The cores generated under this category are:

Loopback This IP core does not perform any image processing operation on the image, instead just returns the input image as output. This core is designed as a *TypeA* core and used mainly for bypassing any image processing operation in complex applications as well as for testing and debugging the DMA transactions in the implementation.

Loopback 2 This IP core is similar to the Loopback core, but is designed as *TypeB* core. Being a *TypeB* core, this core receives two input images, but it ignores the second input and outputs the first one. This core is used when images need to bypass a container meant for *TypeB* cores. The application of this core is more elaborately explained in Section 5.3.2.

Image Subtraction This IP core is used for computing the absolute difference between two images, the result of which is useful for several image processing applications. As this operation needs two input images, the core is designed as a *TypeB* core.

Convert To Gray This IP core converts a colour image to gray scale image. OpenCV software performing similar function in software takes as input a three channel RGB image and converts it to a single channel gray scale image. However, owing to difficulty in outputting single channel images of 8-bit depth through DMA channels designed for handling three channel images of 24-bit depth, the IP core is designed to output three channel image, where the pixel values of all the three channels are same and correspond to the gray scale representation. This core is designed as a *TypeA* core.

Threshold This IP core converts a gray scale image to a binary image. Details of this image processing operation is available in Chapter 5 of [17]. This operation generally works only on single channel images resulting in single channel images as outputs. However owing to the difficulty in handling single channel images of 8-bit depth, this IP core

takes as input a three channel image and applies the thresholding operation on each of the channels and outputs a three channel image.

Thus only two cores, Image Subtraction and Loopback 2, are designed as *TypeB* cores while the rest are *TypeA* cores. All the generated IP cores are constrained to enable them to be clocked with a 100 MHz clock. Complex image processing applications which can be broken down to these smaller image operations can be implemented using partial reconfiguration.

5.1.2 Static and Reconfigurable Logic

This section describes the design of the static and the reconfigurable logic for the implementation of image processing applications using partial reconfiguration. This implementation extends Xillinux hardware design, the files for which can be downloaded from the web page of Xillinux[2]. The hardware design flow utilises XPS for integrating the Video DMA IP cores, the image processing IP cores and other peripheral IP cores with the Xillinux VGA cores. The following IP Cores from Xilinx IP library are used in the implementation:

- Processing System 7 : This core provides an interface around the Processing System of the Zynq-7000 device [51].
- Video DMA : Soft core used for transferring images between memory and the image processing IP cores with high bandwidth [52].
- AXI Interconnects : Used for interconnecting different IP cores using the AXI buses [53].
- AXI General Purpose I/O : Used for interfacing reset signals from the Processing System to the IP cores implemented in the Programmable Logic using AXI4-Lite [54].

The Xillinux hardware implementation utilises two of the four AXI High Performance Ports available for the Programmable Logic to interface with the external memory. Of these two ports, one is used by the hardware FIFOs of the Xillinux and the other by the VGA core to access the frame buffer. However, since the implementation does not need the Xillinux FIFOs, they are disconnected from the AXI High Performance Port and the port is reclaimed. Thus, three AXI High Performance Ports are available for use. The Processing System has four configurable clocks for the Programmable Logic - one of these is configured to 100 MHz and utilised by all the bus interfaces and IP cores integrated into the Xillinux design.

The reconfigurable logic of the implementation consists of three reconfigurable containers of almost equal size. Given the irregular layout of the reconfigurable fabric, restricting all the containers to be of equal size is difficult. Hence one of the three containers, *Container 1*, is slightly larger than the other equal sized containers. The containers are designed to be big enough to accommodate the IP cores which are used as reconfigurable modules and still leave enough resources for implementing the static logic. The resources allocated for the containers are provided in Appendix A.2.

[2]http://www.xillybus.com/download

These reconfigurable containers will undergo frequent reconfigurations and will interface with the static logic consisting of Video DMAs and other peripherals using AXI4 interfaces. Hence, the design has to take sufficient precautions to ensure that there are no pending transactions on the AXI-bus when reconfiguration is in progress. Failure to do so can result in hung AXI buses. Reference design from Xilinx handle this issue by applying reset to the cores inside the containers throughout the reconfiguration process [11]. To achieve this, the design utilises the memory mapped IP core — AXI General Purpose I/O, through which reset can be applied to these containers by the software while executing the reconfiguration process.

With major design issues sorted out, what remains is the selection of AXI4 interfaces to integrate the IP cores. To maximise throughput of Video DMA, it is interfaced with external memory through a High Performance AXI Port between the Processing System and the Programmable Logic using an AXI4-bus. All the IP cores in the design - Video DMA, AXI General Purpose I/Os, Image Processing Cores etc. require configuration for which AXI4-Lite interface is used. For this purpose a single AXI4-Lite bus is used to which the configuration interfaces of all the IP cores are connected. This configuration bus connects to the Processing System through one of the General Purpose AXI Port, giving it the responsibility of configuring all the IP cores in the design. The choice of AXI4-Lite as an interface for configuration is justified by the fact that the volume of configuration data for all peripherals combined is lower than the raw image data to be accessed by the Video DMA. The interfacing of all the cores is shown in Figures 5.2 and 5.3.

The final step for the hardware design is the interfacing of the input and output interfaces for all the containers. The IP cores in these containers can be connected to dedicated Video DMA or the containers can be chained one after the other. Both these designs are described in the following sections.

Independent Containers

This method of connecting reconfigurable containers uses dedicated Video DMA per container making its functioning independent of the other containers. Since Video DMAs are connected to AXI High Performance Ports only, the number of containers that can be created in the design is limited by the availability of these ports. With three free AXI High Performance Ports, only three containers can be connected to Video DMAs. Connection of one VDMA to an image processing core is shown in Figure 4.5, the same is replicated thrice, with the PlanAhead tool designating the image processing cores as reconfigurable modules. To make each container function independent of the other, the reset circuitry developed with the AXI General Purpose I/O also has to be replicated for all the three containers. Thus, during the reconfiguration process of a particular container, only that container is held in reset state while the other containers can continue to function. The hardware design for the scheme of three independently fed reconfigurable containers is shown in Figure 5.2.

The use of one Video DMA per container allows the reconfigurable containers to have only one input AXI4-Stream and one output AXI4-Stream. i.e. only *TypeA* cores can be loaded in these containers. The containers are thus placed in multi-island style, as the used *TypeA* IP

cores can be loaded in any of the containers. Hence, the image processing applications which require the functionality of Image Subtraction cannot be implemented using this scheme.

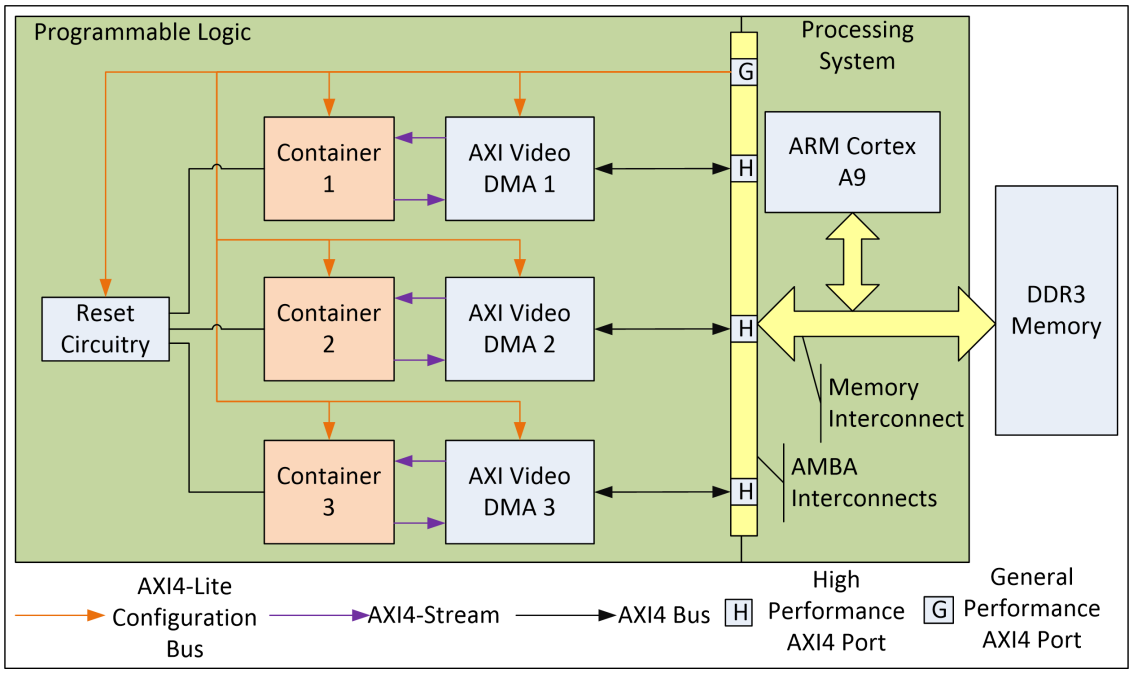


Figure 5.2: Three Independent Reconfigurable Containers

With multiple independent reconfigurable containers for use, multiple image frames can be processed in parallel. The performance of the implemented image processing applications, in terms of the frame rate, depends on the scheduling mechanism used to process the input image frames. Section 5.3 describes the scheduling mechanisms used for implementing different image processing applications. This connection scheme is used for implementing morphological image processing algorithms like *Opening* and *Closing* along with applications like Lane Detection. This design is referred to as *Design with Independent Containers* in the rest of the thesis.

Chained Containers

The connection scheme of independent containers does not support loading of *TypeB* cores which are required for implementing certain image processing operations. For creation of reconfigurable containers with two input AXI4-Streams, two Video DMA cores are required. However, as the number of AXI High Performance ports are limited, all containers cannot be provided with two Video DMA cores for input. Instead, the containers are connected in a chained manner, in which one container feeds its output to the next one connected in the chain. The connection of these containers is shown in the Figure 5.3.

As depicted in the Figure 5.3, *Container 1* has two input AXI4-Streams and can load only cores of *TypeB*, while the other containers can load only the *TypeA* ones. One disadvantage of this connection scheme is that since the containers are connected to each other, they cannot

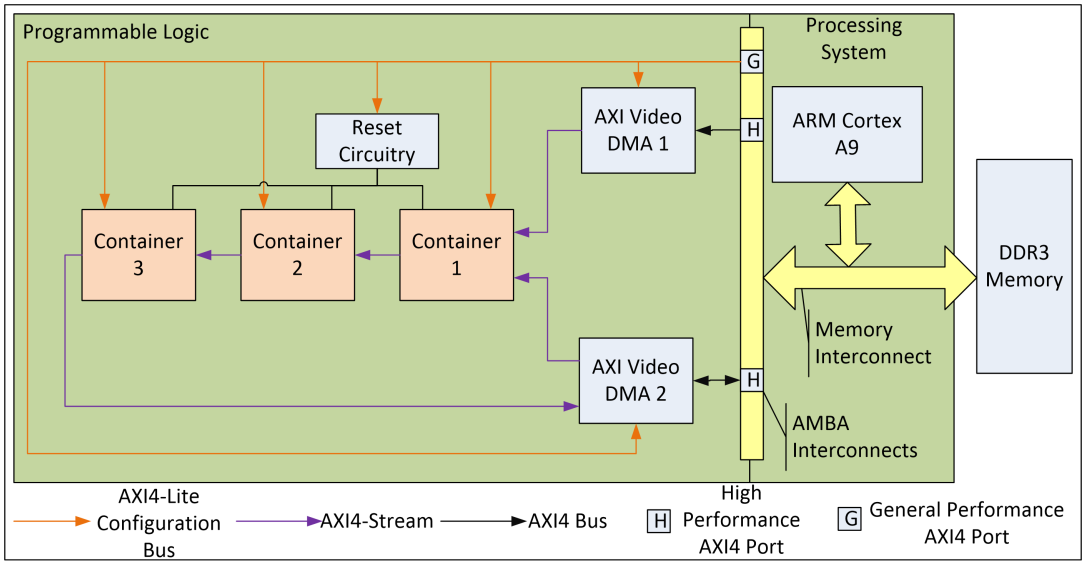


Figure 5.3: Three Chained Reconfigurable Containers

be operated independently. While reconfiguring one of the containers, all the others too must be held in reset state to prevent them from initiating any AXI transactions with the container under reconfiguration. This limits the availability of scheduling options while implementing image processing applications. This connection scheme has been used for implementing applications like Background Subtraction and Morphological Image Gradient. This design is referred to as *Design with Chained Containers* in the rest of the thesis.

5.2 Software Design

The software architecture for implementation of image processing applications follow layered architecture as shown in Figure 5.4. The software needs to interact with hardware components like Video DMA, Processor Configuration Access Port (PCAP) and AXI General Purpose I/O etc., in order to perform partial reconfiguration and control the processing of images. Since all these hardware components are memory mapped, it is possible to utilise the *mmap()* and *munmap()*[3] interfaces from Linux to read from and write to their control registers with programs running at user-space using Algorithms 1 and 2. User-space drivers are simpler to write and debug as compared to the kernel space drivers which provide slightly better performance and system security but are difficult to debug as they require kernel modifications [55]. User-space drivers are also flexible for modifications on account of the implemented functionality. However, misconfiguration of these user-space drivers, e.g. configuring Video DMA to overwrite memory controlled by Linux kernel, can be disastrous and lead to system crashes. Despite this security flaw, user-space drivers are used to reduce development effort.

[3]Information about the functions mmap() and munmap() available in Linux man pages.

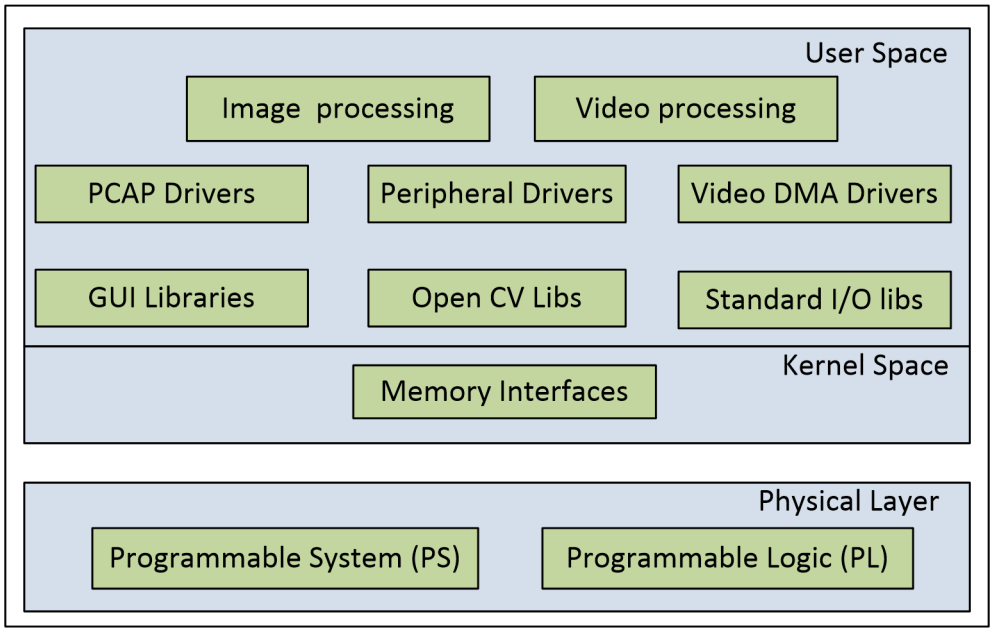


Figure 5.4: Software Architecture for the Prototype

The slight performance penalty is accepted in return for the above mentioned flexibility. To prevent mishaps, the application software is designed to perform sufficient feasibility checks.

Algorithm 1 Reading from Device Registers

1: **procedure** *GetRegister*(Offset)
   {Feasibility check to verify if the address being read is within the device}
2: **if** $(Offset < OffsetMax) \ \wedge (Offset >= OffsetMin)$ **then**
3: $devicePtr \leftarrow mmap(deviceAdr, range)$
4: $Value \leftarrow devicePtr[Offset]$
5: $munmap(devicePtr)$
6: $return(Value)$
7: **end if**
8: **end procedure**

As shown in Figure 5.4, the image and video processing applications form the topmost layer in the architecture. These applications use the lower layer modules like the user-space drivers for interacting with the hardware, the High GUI (described in Section 3.3.1) for displaying the images after processing, the Open CV libraries for utilising the abstractions of image data and the other standard C I/O libraries for printing to the console and acquiring user input.

Of the 512MB DDR3 memory available on board, 450 MB is reserved for the user-space and the kernel space of Linux, while the remaining memory is designated for storing the partial bitstreams of the image processing IP cores and the image frames being processed. Figure 5.5 shows the memory organisation of the implementation. The memory area allocated for partial bitstreams is divided into bitstream slots, each slot of size 300KB. The size of partial

Algorithm 2 Writing to Device Registers

1: **procedure** *SetRegister*(Offset, Value)
{Feasibility check to verify if the address being written is within the device}
2: **if** $(Offset < OffsetMax) \ \wedge (Offset >= OffsetMin)$ **then**
3: $devicePtr \leftarrow mmap(deviceAdr, range)$
4: $devicePtr[Offset] \leftarrow Value$
5: $munmap(devicePtr)$
6: **end if**
7: **end procedure**

bitstreams of all the used IP cores, as shown in Appendix A.2, does not exceed 300KB. Similarly, the memory area for image frames are divided into smaller image slots, each of 1MB. The images used for implementation in this thesis have a resolution of 640 x 480 pixels with 24-bit depth, which is approximately 900KB. Boundary checks make sure that Video DMA transactions lie within the memory region meant for image frames and PCAP transactions lie within the region meant for bitstreams.

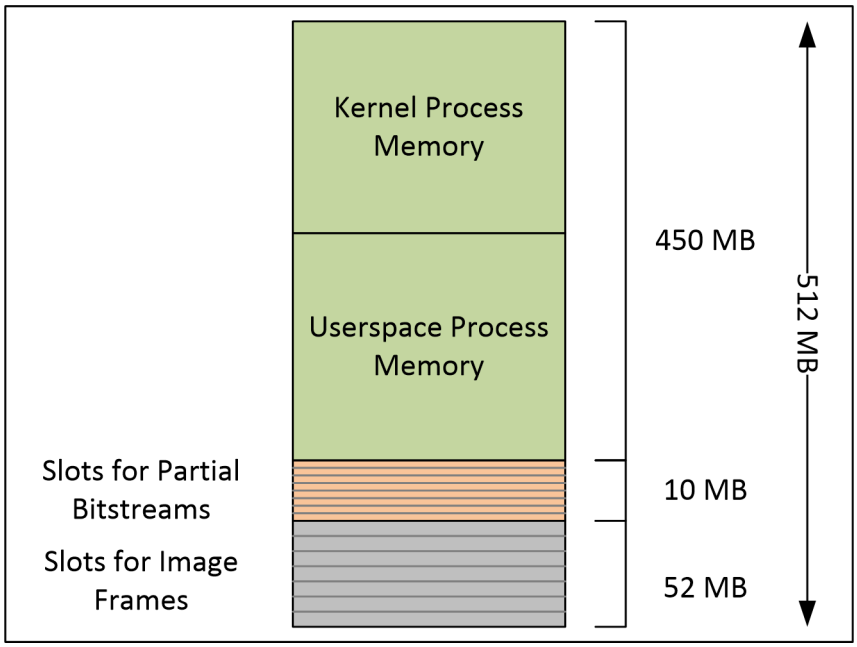


Figure 5.5: Memory Organisation for the Prototype

The application software implementing any image processing or video processing algorithm starts with a one time configuration of the AXI High Performance ports which interface with the used Video DMAs. The image frames are acquired from a camera or a video and stored in the image frame slots. The reconfigurable containers are loaded with IP cores responsible for the desired image processing operations and the cores are executed with the corresponding Video DMAs configured to feed them the input images. The processed images returned by the containers are stored in the image frame slots and later displayed using the GUI libraries.

Algorithm 3 Scheduling of IP cores for parallel execution using three independent containers

```
1: procedure Main
   {Software for controlling Partial Reconfiguration}
2: Configure AXI-HP ports to 64-bit width to interface with Video DMA
3: Reset all the Video DMAs
4: while true do
5:    Query for image frame from camera/video
6:    if IP core in Container 1 is to be changed then
7:       Assert reset signal for Container 1
8:       Perform reconfiguration and wait till it is over
9:       Deassert reset signal for Container 1
10:   end if
11:   Execute Container 1 and its Video DMA
12:
13:   Query for image frame from camera/video
14:   if IP core in Container 2 is to be changed then
15:      Assert reset signal for Container 2
16:      Perform reconfiguration and wait till it is over
17:      Deassert reset signal for Container 2
18:   end if
19:   Execute Container 2 and its Video DMA
20:
21:   Query for image frame from camera/video
22:   if IP core in Container 3 is to be changed then
23:      Assert reset signal for Container 3
24:      Perform reconfiguration and wait till it is over
25:      Deassert reset signal for Container 3
26:   end if
27:   Execute Container 3 and its Video DMA
28:
29:   while Container 1 is not idle do
30:      wait
31:   end while
32:   Display output of Container 1
33:
34:   while Container 2 is not idle do
35:      wait
36:   end while
37:   Display output of Container 2
38:
39:   while Container 3 is not idle do
40:      wait
41:   end while
42:   Display output of Container 3
43: end while
44: end procedure
```

The basic sequence of operations for processing three image frames in parallel using three independent containers, using one container per frame, is given in the Algorithm 3. Based on the number of containers used, the algorithm must be modified. Applications can appropriately modify the algorithm for reconfiguring the containers with desired cores and then using it as an image processing pipeline with the image frame slots in memory behaving as pipeline registers. In such a case, the input image is fed to Container 1, while the output of *Container 1* is given to *Container 2* and so on. The final processed image is obtained from the container which is the last stage of the pipeline.

While using the hardware design with chained containers, the above mentioned algorithm needs to be modified for performing partial reconfiguration of all three containers one after the other while all of them are held in reset state. The input images are fed to *Container 1* while the processed images are obtained from *Container 3*. Due to the design, parallel processing of images using multiple containers is not possible.

The application software currently polls to check the completion of the image processing operations and the reconfiguration process. Instead of this polling, interrupt based mechanisms can be used. However Linux does not provide interfaces for overriding or implementing interrupt service routines in user-space. Theoretically, it is possible to implement a generic interrupt service routine in kernel space and intimate user-space processes by means of signals. However, considering the complexity brought along by modifications of Linux kernel, this thesis uses polling in the current implementations.

The next section describes all the different image processing algorithms implemented with the hardware utilising the three independent containers or the three chained containers along with the schedule used for partial reconfiguration.

5.3 Scheduling of Partial Reconfiguration

Chapter 3 introduced different image processing applications which provide good use cases for study of partial reconfiguration. This section deals with the implementation of those image processing algorithms by means of software based static schedules for loading the image processing IP cores into the reconfigurable containers. Applications have been implemented with different schedules, in order to evaluate the impact of these software schedules on their performance.

5.3.1 Morphological Image Processing

Opening and Closing of Images

The morphological image operations, *Opening* and *Closing*, are implemented using the Dilation and the Erosion IP cores with a two-step schedule. These operations are implemented using the hardware design with three independent containers, explained in Section 5.1.2. The Table 5.1 presents the schedule for the implementation of a single iteration of *Opening* and

Closing using all the three available containers. In this schedule, each container processes one image frame completely, thus enabling parallel processing of three image frames to provide a better throughput. The resulting image of the first step is given as input to the same container at the second step. The output of the second step is the resulting processed image. The execution of this schedule is shown in Figure 5.6.

	Opening			**Closing**		
	Cont. 1	**Cont. 2**	**Cont. 3**	**Cont. 1**	**Cont. 2**	**Cont. 3**
Step 1	Erosion	Erosion	Erosion	Dilation	Dilation	Dilation
Step 2	Dilation	Dilation	Dilation	Erosion	Erosion	Erosion

Table 5.1: Schedule for Morphological Image Processing - Opening and Closing

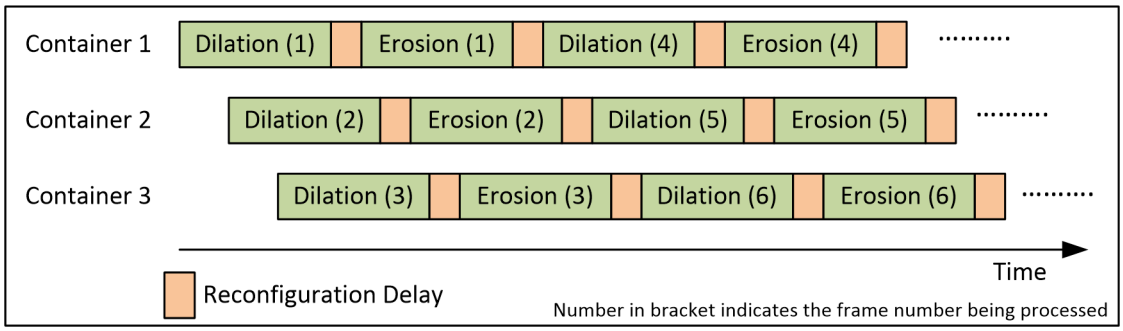


Figure 5.6: Execution of single iteration of Closing using three reconfigurable containers

For multiple iterations of the operations, each step of the schedule is repeated as many times as the number of iterations desired. This is different from the conventional idea of iterations where the entire schedule is repeated. This is briefly explained in Section 3.2 of this thesis and in Chapter 5 of [17]. The execution of the schedule for performing n-iterations of the *Closing* operation is shown in Figure 5.7. It can be observed that the number of reconfigurations per processed frame remains constant for the operation irrespective of the number of iterations performed.

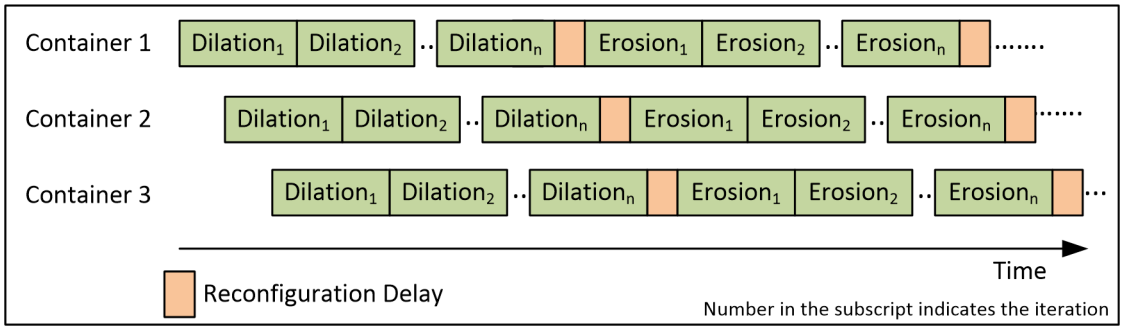


Figure 5.7: Execution of n-iteration of Closing using three reconfigurable containers for a single image frame

These operations can also be implemented using static hardware, where one container is preloaded with the Dilation IP core and the other with the Erosion IP core. These can then work in tandem to achieve the functionality of *Opening* or *Closing*. Such an implementation does not use partial reconfiguration. The Figure 5.8 shows computation of a single iteration of *Closing* using the described static hardware. This implementation can also be used to perform *Opening* or *Closing* iteratively by appropriately using the preloaded cores to achieve multiple dilations and erosions.

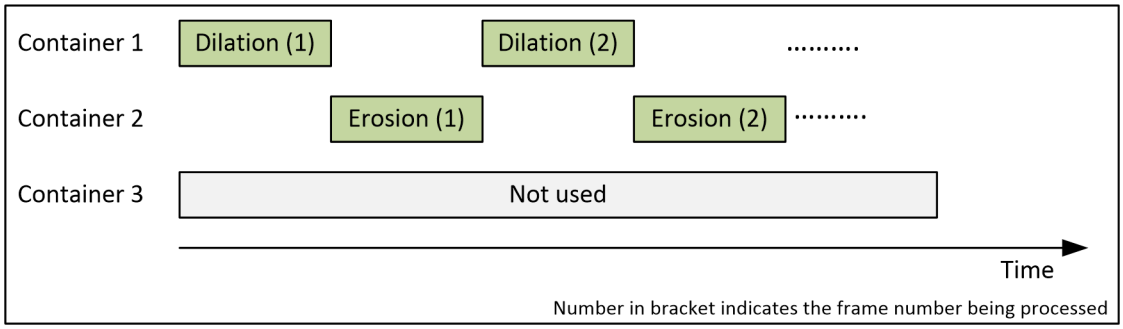


Figure 5.8: Execution of single iteration of closing using static hardware

The evaluation results for this application with all these different schedules are presented in Section 6.2.1.

Morphological Gradient

Like the morphological image processing applications of *Opening* and *Closing*, Morphological Gradient is also implemented using the Dilation and the Erosion IP cores. However, as mentioned in Section 3.2, Morphological Gradient involves subtracting two images. Hence, it can be implemented only on the hardware design with chained containers. This is because the IP core for subtracting two images can only be loaded in the *Container 1* of the design with chained containers, described in Section 5.1.2. The schedule for the implementation of the application is given in Table 5.2.

For the single iteration of this application, the input image is fed to *Container 1* during Step 1. The resulting dilated image is stored in the memory and the input image is again given to *Container 1* after reconfiguration for performing Step 2. The resulting eroded image is also stored separately in the memory. The dilated image, from Step 1, and eroded image, from Step 2, are given to *Container 1* for subtraction in the next step to get the morphological gradient of the input image. The execution of this schedule for one iteration is shown in Figure 5.9.

For multiple iterations, multiply dilated and eroded images need to be stored separately and passed as input to the Image Subtraction core. The schedule presented in Table 5.2 must be modified according to the number of iterations, just as in the case of the other morphological image operations — *Opening* and *Closing*.

	Single Iteration			Three Iterations		
	Cont. 1	**Cont. 2**	**Cont. 3**	**Cont. 1**	**Cont. 2**	**Cont. 3**
Step 1	Bypass	Dilation	Bypass	Bypass	Dilation	Dilation
Step 2	Bypass	Erosion	Bypass	Bypass	Dilation	Bypass
Step 3	Subtraction	Bypass	Bypass	Bypass	Erosion	Erosion
Step 4				Bypass	Erosion	Bypass
Step 5				Subtraction	Bypass	Bypass

Table 5.2: Schedule for Morphological Image Processing - Morphological Gradient

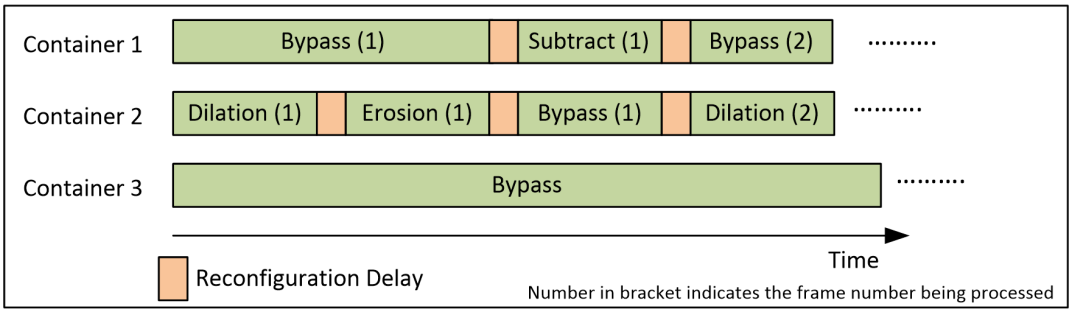


Figure 5.9: Execution of single iteration of Morphological Image Gradient

Unlike in *Opening* and *Closing*, for Morphological Gradient the number of reconfigurations per processed frame depends on the number of iterations to be performed. The number of reconfigurations required for different number of iterations of Morphological Gradient is given in the Table 5.3. It can be seen that performing the operation iteratively thrice requires more reconfigurations as compared to performing it four times. This anomaly is attributed to the hardware design with the chained containers and the developed schedule for reconfiguration. Its impact on the performance of the implementation is presented in the Section 6.2.1.

No.of Iterations	**No. of Reconfigurations**
1	5
2	8
3	9
4	8

Table 5.3: Number of reconfigurations per processed image - Morphological Gradient

5.3.2 Background Subtraction

As explained in Section 3.2, this thesis implements Frame Differencing, a method of background subtraction which can indicate regions of motion in an image frame. This application requires capability of subtracting an image from another and is hence implemented using the hardware design with chained containers. As explained in Section 3.2, this application takes

the image frames of a continuous video as input and outputs the edges of moving objects in those frames. The required IP cores to implement this application are Image Subtraction, Threshold, Convert To Gray. The background subtraction is performed in two steps, as shown in Table 5.4.

	Container 1	**Container 2**	**Container 3**
Step 1	Bypass	Convert To Gray	Bypass
Step 2	Image Subtraction	Threshold	Bypass

Table 5.4: Schedule for Background Subtraction

In the first step, the input image is fed to *Container 1*, which simply feeds it to *Container 2*. At the end of Step 1, gray scale image of the current frame is available from *Container 3* which is stored in the memory. In the second step, this gray scale image is subtracted from the gray scale image of the previous frame which was also stored in the memory. The computed difference image is given as input to the Threshold core (configured in *Container 2*) in the same step, which enhances the difference if it is above a pre-defined threshold. The execution of this schedule is shown in Figure 5.10.

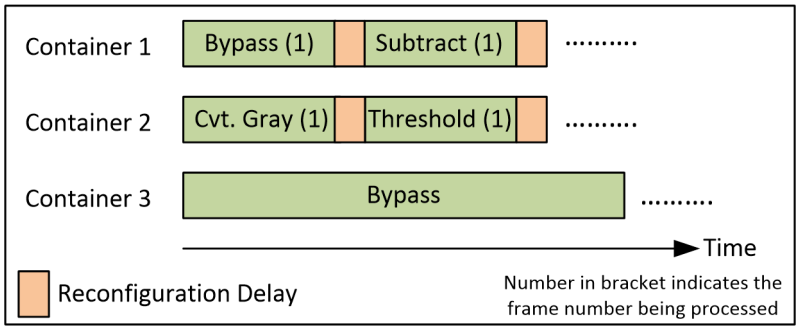


Figure 5.10: Execution of Background Subtraction using partial reconfiguration

Section 6.2.2 presents the performance of this implementation against its software counterpart executed on the Processing System.

5.3.3 Lane Detection

Lane detection, as explained in Section 3.2, can be implemented using partial reconfiguration if hardware implementations of all the involved operations are available. However, Hough Transform, the last step of the algorithm which detects the edges representing the lanes, could not be synthesized successfully to fit into the designed reconfigurable containers. Attempts to reduce the size of the core by reducing the size of images which can be processed were not fruitful. Hence, the thesis implements only edge detection on the images. This implementation can be extended to detect lanes on availability of IP cores of a suitable size for performing the Hough Transform.

The sequence of operation for the edge detection is as follows : Conversion to Gray Scale - Noise filtering - Sobel Operator for edge detection - Thresholding - Morphological Opening. This sequence returns edges of the input image and can be overlaid on the input image for marking the edges. The overlaying process may be done in software resulting in a image processing pipeline implemented partially in software and partially in hardware. The thesis does not investigate this as software/hardware partitioning is out of the scope of this thesis.

Edge detection can be implemented with five IP cores, one for each processing step. This image processing pipeline for edge detection can be implemented for processing three image frames in parallel, like in Morphological Opening and Closing. We can call such a schedule as *One Frame per Container* schedule. The execution of such a schedule is shown in Figure 5.11.

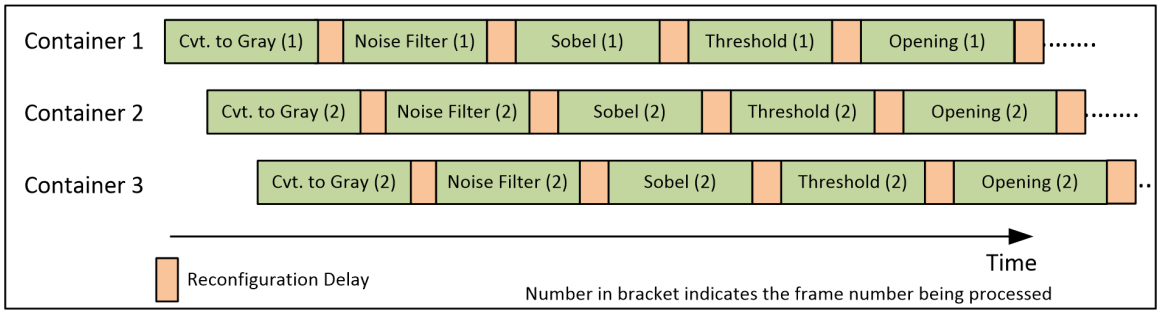


Figure 5.11: Execution of Edge Detection using the schedule — One Frame per Container

Edge detection can also be implemented with a schedule for minimising reconfigurations as described in Table 5.5. The execution of this schedule is shown in Figure 5.12. This schedule reduces the number of reconfigurations required, mainly because *Container 3* is dedicated for performing the Sobel operation. In this schedule, all the containers wont be executed in parallel. The input image is channelled through all these IP cores in sequence. i.e. at a given point of time only one container is processing the image, while the other containers are free. We call this schedule as *One Frame at a Time*.

	Container 1	**Container 2**	**Container 3**
Step 1	Convert To Gray	Noise Filter	Sobel
Step 2	Threshold	Opening	-

Table 5.5: Schedule for Edge Detection — One Frame at a Time

This schedule can provide better performance if the implementation deviates from the Algorithm 3. Instead of performing reconfigurations only when the containers with incorrect IP cores are detected (lines 6, 14 and 22 of Algorithm 3), the scheduler can anticipate the upcoming reconfigurations and perform them when some other container is processing the image. This allows the reconfiguration delay to be subsumed within the execution time of the containers. The subsumption of the reconfiguration delay within the execution periods of the container is shown in Figure 5.13.

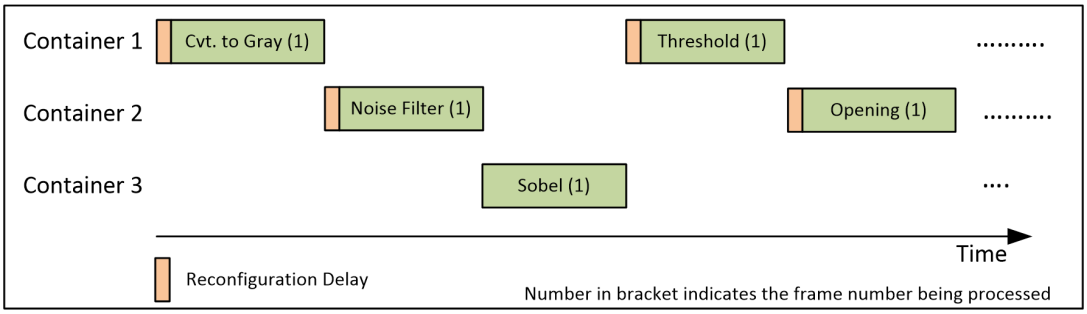


Figure 5.12: Execution of Edge Detection with the schedule — One Frame at a Time

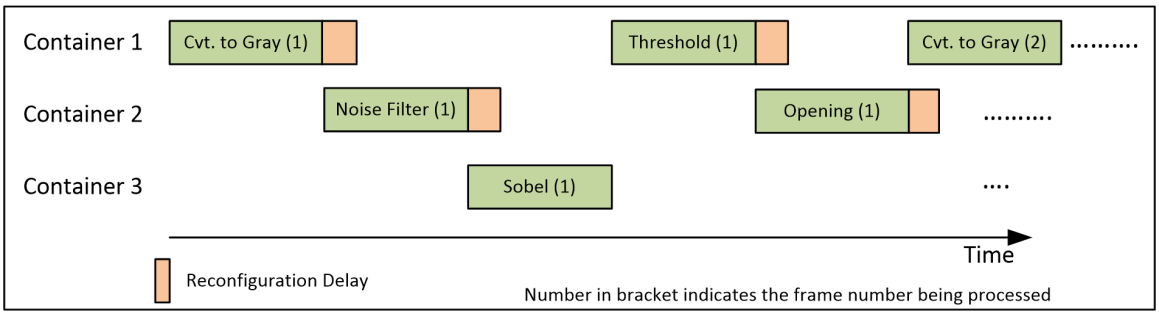


Figure 5.13: Execution of Edge Detection under schedule — One Frame at a Time after optimisation

The performance of Edge Detection under all the described schedules is presented in Section 6.2.3.

Summary

Implementing image processing applications on Zynq-7000 platform requires the availability of many image processing operators to be available as IP cores. Image processing operators can be generated as IP cores using the Vivado High Level Synthesis Tool and the HLS Video Library. As a part of the thesis, IP cores for different image processing operations were generated. These IP cores can be loaded in three reconfigurable containers, created for implementing image processing applications. As the interfaces of the IP cores must match the interfaces of the reconfigurable containers where it is loaded, a particular IP core can be loaded only on a given set of containers. Software based schedules are responsible for loading appropriate IP cores in the containers for implementing the image processing applications. Different image processing applications can be implemented on this platform by modification of this software based schedule. This thesis implemented Morphological Image Processing applications along with applications like Edge Detection and Background Subtraction with various schedules.

Chapter 6

Results

Contents

This chapter presents the results of different evaluations performed on the implemented image processing applications. The evaluation strategy was mainly aimed at measuring the resource savings and the computational speed-up that can be achieved using partial reconfiguration.

6.1 Resource Consumption with Partial Reconfiguration

The Artix-7 reconfigurable fabric, which makes up the Programmable Logic of the used Zynq-7000 device, mainly consists of Configurable Logic Blocks (CLBs), DSP Slices, FIFO slices etc. This section summarises all the resources available in the Zynq-7000 device used in the Zedboard and the usage statistics of the implemented applications.

Each CLB in the reconfigurable fabric of the Zynq-7000 device, XC7Z020-CLG484-1, consists of two slices. Each CLB slice is made up of four 6-input look-up tables (LUTs), eight flip-flops, multiplexers and arithmetic carry logic. One-third of the total slices, known as *SliceM*, can use their LUTs as distributed RAM or as shift registers. The rest of the slices are known as *SliceL*. The lookup tables from these slices are used for implementing combinational or sequential logic of the design. The DSP blocks are needed for performing complex mathematical operations while the RAM is used to store data. More information on the structure of slices and the reconfigurable fabric is provided in [56]. The total available resources in the used Zynq-7000 device are given in Table 6.1.

	Availability	Requirements			
Resource	**Zedboard**	**Dilation**	**Erosion**	**Opening**	**Closing**
Registers	106400	1529	1618	2647	2647
Look up tables	53200	1783	2097	3406	3406
SliceL	8950	241	288	484	484
SliceM	4350	206	237	368	368
DSP Slices	220	0	0	0	0
RAM - 18Kb Blocks	280	4	4	9	9

Table 6.1: Resources - Availability in Zedboard and Requirements for Morphological Image Operations of Opening and Closing

6.1.1 Morphological Image Processing

Opening and Closing

The implementation of Morphological Image Operations — *Opening* and *Closing* — by reconfiguring the implemented containers with IP cores for Dilation and Erosion have been elaborately described in Section 5.3.1. As compared to using a static IP core which performs the entire operation, using partial reconfiguration provides resource savings as shown in Table 6.1.

Among the two IP cores, the Erosion core consumes more resources than the Dilation core. Container sizes big enough for the Erosion core would also suffice for Dilation. The approximate resource savings are summarised in Table 6.2. These figures are approximate as they do not consider the overhead resources required for routing, as explained in Chapter 2.

Resources	**With Partial Reconfiguration**	**Without Partial Reconfiguration**	**% Savings**
Registers	1618	2647	38.43%
Look up tables	2097	3406	38.87%
SliceL	288	484	40.49%
SliceM	237	368	35.59%
DSP Slices	0	0	0
RAM - 18Kb Blocks	5	9	44.44%

Table 6.2: Resources Savings in Morphological Image Operations of Opening and Closing

Morphological Gradient

Morphological Gradient has been implemented with three image processing IP cores — Dilation, Erosion and Image Subtraction — on the hardware design with three chained containers.

The implementation of this application and the corresponding schedule is described in Section 5.3.1. With partial reconfiguration, the effective required FPGA resources is the sum of the resources consumed by the Image Subtraction core and the bigger of the Dilation and the Erosion core, i.e. res(Image Subtraction) + max (res(Dilation), res(Erosion)); where res() is the resource consumption of the IP core. Resource savings are evaluated in this manner because the FPGA resources are multiplexed only between the Dilation and the Erosion IP cores.

The approximate resource savings in implementing the application of Morphological Image Gradient is presented in Table 6.3. The resource savings for this application have not been as substantial as for *Opening* and *Closing*.

Resources	**With Partial Reconfiguration**	**Without Partial Reconfiguration**	**% Savings**
Registers	3425	3590	4.59%
Look up tables	2483	2738	9.31%
SliceL	454	508	10.62%
SliceM	404	390	-3.58%
DSP Slices	0	0	0
RAM - 18Kb Blocks	5	9	44.44%

Table 6.3: Resources Savings in Morphological Image Gradient

The extent of resource savings on account of partial reconfiguration based design depends on the application as well as the manner in which the application is decomposed into smaller operations. Finer granularity of the operations potentially leads to more resource savings, as the container size needs to be just big enough to accommodate the largest of the operations.

6.2 Processor Speedup with Partial Reconfiguration

As explained in the Chapter 5, the Processing System runs Xillinux, a Linux distribution, as the operating system. The software program, responsible for scheduling the reconfiguration process, executes in the user-space of the Linux environment. For evaluating the performance of the image processing applications, the time between the start and the end of the image processing is measured. This excludes the time required for acquiring the input images from a video, a camera or any other source. The time required for displaying them in a GUI window is also excluded. The time is measured by noting the timestamps using the function *clock()*[1] which provides an approximation of the current processor time in terms of clock cycles. The processing time for an image frame is calculated by dividing the difference between the timestamps at the start and at the end of the frame processing by the total number of processor cycles in a second.

[1]Linux man pages - http://linux.die.net/man/3/clock

It is important to note that this value is only approximate and influenced heavily by the context switch times. In general the measured timings have been observed to deviate by around 5 ms. To mitigate the effect of these deviations, all times have been measured as an average of processing times of at least 150 frames. For measurement purposes the scheduler is programmed to process 200 images of which the initial 50 images are not considered for measurement. This gives time for any housekeeping processes spawned by Linux on account of the scheduler program to settle down. The scheduler measures the processing times for the next 150 image frames and reports the average processing time per frame.

6.2.1 Morphological Image Processing

Opening and Closing

In order to evaluate the processor speed-up for *Opening* and *Closing*, the performance of the application is measured when executed with different levels of hardware usage. The different implementations of the application are enumerated below :

- Case A : Complete software implementation using OpenCV executing on the Processing System.
- Case B : Single hardware accelerator in the Programmable Logic for performing the entire operation. The hardware accelerator is loaded in one of the reconfigurable containers. Hence, partial reconfiguration is not used.
- Case C : Use of hardware accelerators for Dilation and Erosion. The cores are loaded into the containers based on a pre-determined schedule as explained in Section 5.3.1. This use case is implemented using one, two and three reconfigurable containers, where each container is responsible for processing one image frame.
- Case D : Loading Dilation core in one container and Erosion core in another. Chaining the input frame through these containers to achieve the *Opening* and *Closing* operation. No runtime partial reconfiguration is required in this case.

The performance of each of these implementations for executing different number of iterations of *Opening/Closing* operations was measured. The process for iteratively performing *Opening* and *Closing* operations on an input image is described in Section 5.3.1 of this thesis. The results of the evaluation are summarized in Table 6.4.

The results suggest that the fastest implementation for performing Morphological Opening or Closing is the one using a single hardware accelerator to perform the entire operation (Case B). However, such an approach is inflexible as it cannot be used for performing the operation iteratively on the input image. Also, Section 6.1.1 presented that such an approach consumes more FPGA resources.

The speed-up comparison for the hardware implementations (Case C and Case D) are presented in the Figure 6.1. The static hardware in the figure refers to the Case D of this application. It can be seen very clearly that hardware support provides significant speed-up

	Processing time required per frame in ms					
	Case A	**Case B**	**Case C**			**Case D**
Iterations	**Software**	**Single HW Module**	**One Container**	**Two Containers**	**Three Containers**	**Static Hard-ware**
1	89.2	3.9	13.7	9	7.8	8.4
2	139.3	NA	22.2	13.1	11.2	15.9
3	171.7	NA	29.8	16.9	14.4	23.8
4	225.3	NA	38.2	20.6	16.5	31.7

Table 6.4: Performance of Morphological Image Operations - Opening and Closing

for this image processing application. Using just a single accelerator provides a speed-up exceeding 5x. The irregularity in the trends of speed-up with increasing complexity of the application could not be explained. It may be attributed to the experimental set-up where measurement times are influenced by the processor load on account of background processes and context switch times of Linux.

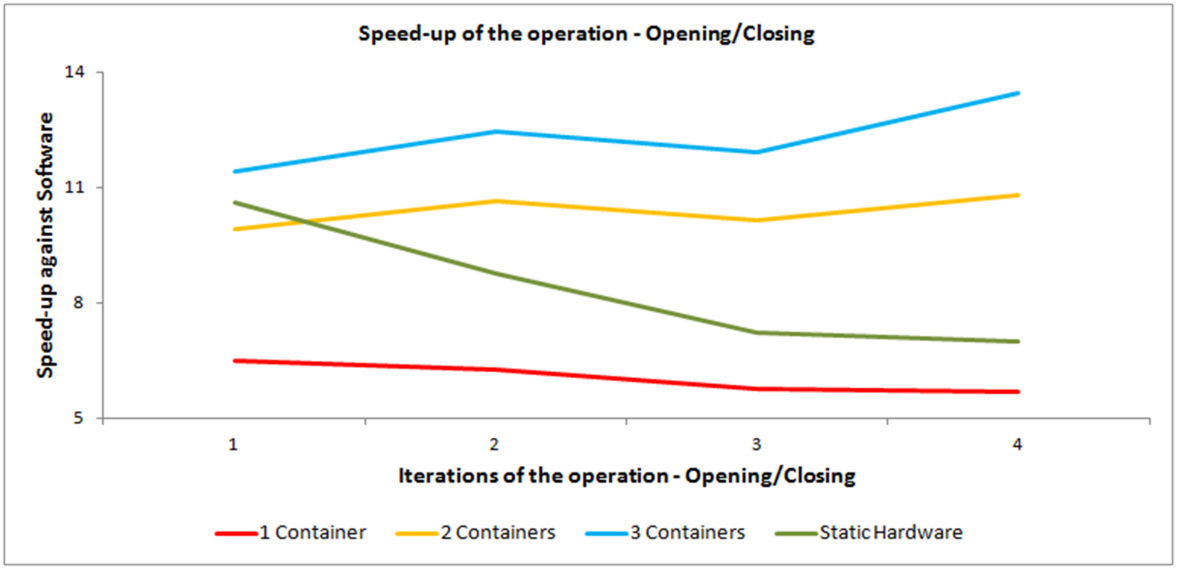


Figure 6.1: Speed-up Comparisons for Opening/Closing

The partial reconfiguration based approach using the reconfigurable containers (Case C) adds to the overheads but is still significantly faster than the software implementation (Case A). Using multiple containers for processing multiple images in parallel reduces processing time per frame as shown in Table 6.4.

For implementing *Opening* or *Closing* using partial reconfiguration, each used reconfigurable container is configured twice per processed image. This reconfiguration adds an additional

overhead. This overhead can be observed by comparing the frame processing times of the implementation which uses partial reconfiguration on one container in Case C with implementation using preloaded Dilation and Erosion IP cores in Case D, as shown in Figure 6.2.

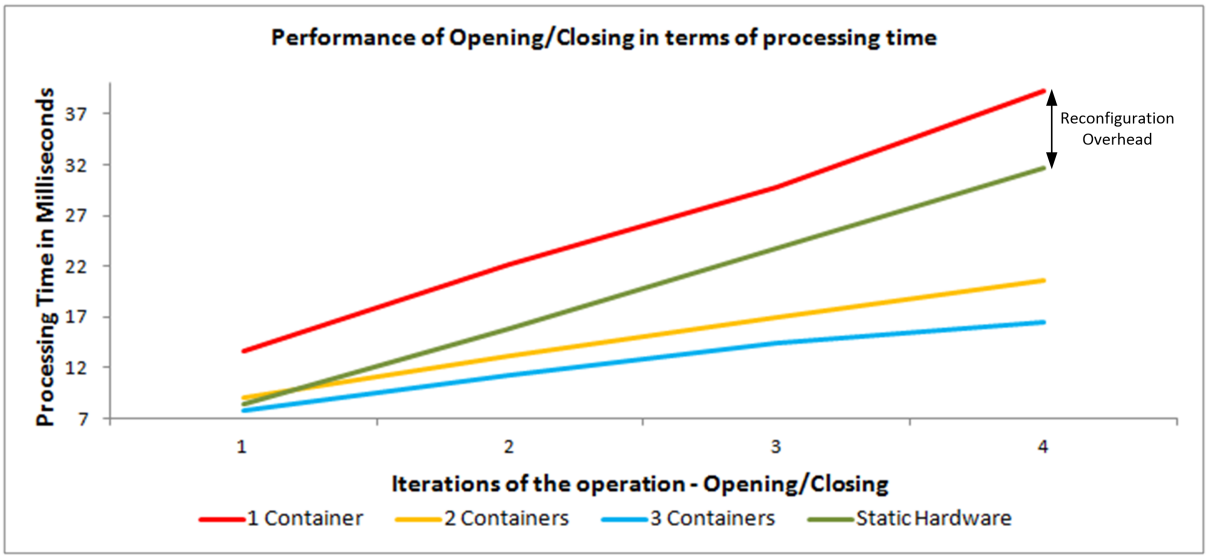


Figure 6.2: Processing times for Opening/Closing with different implementations

Both the implementations differ only by the fact that the used reconfigurable container is reconfigured twice per processed image frame in Case C while no reconfigurations are performed in Case D. As shown in Table 6.4, the processing times in Case D has a constant offset of approximately 6ms with the times in Case C using one container. This constant offset is the time lost in the two reconfigurations. This constant offset between the two cases is clearly visible in Figure 6.2.

Morphological Image Gradient

For Morphological Image Gradient, the different implementations that are available are

- Software implementation using OpenCV executed by the Processing System,
- Single hardware accelerator in the Programmable Logic performing the entire operation,
- Partial reconfiguration based schedule using Erosion, Dilation and Image Subtraction IP cores described in Section 5.3.1.

The performance of these implementations for computing Morphological Gradient iteratively is evaluated. The evaluation results of these implementations are summarized in Table 6.5.

Similar to the evaluation results of *Opening* and *Closing*, here as well the implementation with a dedicated hardware accelerator provides the best performance. But the implementation cannot be used to iteratively perform this operation.

Iterations	Software	Single Hardware Module	Partial Reconfiguration based Hardware
1	106.6	4.1	25.4
2	149.3	NA	31.7
3	183.3	NA	44
4	232.8	NA	41.2

Table 6.5: Processing time per frame (in ms) for Morphological Image Gradient

The implementation using partial reconfiguration provides a significant speed-up as compared to its software counterpart and is also flexible enough for performing the operation iteratively.

As presented in Section 5.3.1, the number of reconfigurations performed per processed image depends on the number of times the operation is performed. Table 5.3 shows that the number of reconfigurations for performing this operation iteratively thrice is higher than performing it four times. The impact can be seen in the processing times of the operation where performing the operation thrice consumes more time as compared to performing it four times.

6.2.2 Background Subtraction

The evaluation of Background Subtraction was done by comparing only two implementations - Software implementation using OpenCV and the partial reconfiguration based implementation presented in the Section 5.3.2 of the thesis.

	Background Subtraction with Thresholding	Background Subtraction without Thresholding
Software	13.6	11.9
Hardware with Partial Reconfigurations	18.0	18.0

Table 6.6: Processing times (in ms) for Background Subtraction

This implementation seems to contradict the conclusions from the earlier evaluations that partial reconfiguration based design speeds up computations. However the explanation for this lies in the operations involved in Background Subtraction - Conversion to gray scale, image subtraction and thresholding. The execution of these operations in software is only a little slower than the corresponding applications in hardware, as shown in Appendix A.3. This is mainly because all these image operations operate on a single pixel and not its neighbourhood. Also the Background Subtraction is implemented using the implementation with chained containers with the schedule presented in Table 5.4. The bypassing of the containers adds to the overhead of entire operation and slows down the implementation.

The schedule presented in Table 5.4 implements background subtraction with thresholding. The thresholding is a distinct operation in the software and hence consumes some time as seen in the results. However, if thresholding need not be performed in the hardware implementation, then the Bypass core must be loaded instead of the Threshold core. The time required for execution of Bypass or Threshold IP core is almost the same, hence the execution times are similar irrespective of whether thresholding is performed or not.

6.2.3 Edge Detection

As explained in Section 5.3.3, the Lane Detection application could not be implemented as the IP core for performing the Hough Transform could not fit into any of the reconfigurable containers. Hence, the implemented application detects only edges and not lanes. The application is evaluated by comparing the performance of its Open CV based software implementation against the two implementations with partially reconfigurable schedules, *One Frame per Container* (OFC) and *One Frame at a time* (OFT), as described in Section 5.3.3. The results of the evaluation are summarised in Table 6.7.

	Software	**OFC 1 container**	**OFC 2 containers**	**OFC 3 containers**	**OFT**	**OFT (opt)**
Processing time per Frame (in ms)	113.4	31.5	27.4	17.2	28.3	20.1

Table 6.7: Performance of Edge Detection

The performance of the three implementations are evaluated for detecting edges. As anticipated, the implementations using hardware provide tremendous speed ups.

Like in the case of Morphological Opening and Closing operations, here as well it can be seen that processing time per frame reduces with an increase in the number of containers used for processing the image frames in parallel.

As discussed in Section 5.3.3, the schedule *One Frame at a time* reduces the number of reconfigurations from 5 in case of *One Frame per container* to 4 by dedicating a container for performing a specific operation. The impact of the reduced number of configurations is visible when the processing times for *One Frame at a time* and *One Frame per Container* using 1 container are compared. The processing time per frame using the schedule *One Frame at a time* needs about 3ms lower than the other schedule. It gains this time purely on account of the reduced number of reconfigurations.

Finally, it is possible to optimise the implementation using the schedule *One Frame at a time* by deviating from the Algorithm 3. By trying to subsume the reconfiguration overhead within the execution of the containers, as explained in Section 5.3.3, the processing time per frame

can be reduced. The Table 6.7 presents the performance with this optimised implementation of the schedule *One Frame at a time* under the heading OFT(opt).

6.3 Power Savings with Partial Reconfiguration

Chapter 2 presented power savings as one of the benefits of using partial reconfiguration based design. Clearly, by being able to implement hardware designs on smaller FPGA architectures and by unloading hardware modules during idle phases, power consumption can be reduced. However, the reconfiguration process introduces some energy overheads which cannot be simply neglected. The important question is if the energy overheads introduced by partial reconfiguration is amortized by savings on account of it. This thesis used the Processor Configuration Access Port (PCAP) for reconfiguration in a processor centric manner. The measurement of the power consumption by the PCAP on account of partial reconfiguration was hindered by the lack of power models for the used Zynq-7000 device. However, a similar study on Xilinx Virtex-4 architectures showed a significant reduction in total energy consumption after accounting for energy consumed for partial reconfiguration by an Internal Configuration Access Port (ICAP) [57]. Hence, it would be reasonable to expect that the energy consumed by the PCAP for partial reconfiguration would be lower than the energy savings achieved.

The power consumption of the hardware accelerators (image processing IP cores) developed as a part of this thesis is provided in Appendix A.4. Each of the accelerators consume on an average 1mW. The software executing on the ARM core with its NEON co-processors and the complex memory hierarchy consumes much more power. The average power consumption of the ARM cores in question has been estimated in few unpublished sources[2] to be around 500mW. While this statistic could not be verified, it sounds plausible.

With the hardware implementations which use partial reconfiguration, large fractions of the ARM cores and its complex memory hierarchy are not required to improve performance. The software scheduler executed on the ARM core only configures the DMA for bitstream and image transactions along with polling to find the status of the transactions. The scheduler does not benefit much from the memory hierarchy of the ARM core. Switching off the unused portions of the ARM cores, like the L2 cache, would not penalise the scheduler in terms of the performance. With this step, significant power savings can be expected. However, the behaviour of the implementations with L2 cache of the ARM core switched off could not be evaluated owing to difficulties in switching off the L2 cache which required modifications in Linux kernel.

[2]Presentation slides from Department of Computer Science - University of Virginia: http://www.cs.virginia.edu/~skadron/cs8535_s11/ARM_Cortex.pdf

Summary

This chapter presented the results of the evaluations performed on the image processing applications implemented using partial reconfiguration. The evaluation strategy measured the resource savings and the computational speed-up for the implemented applications. The results showed that implementations using partial reconfiguration provides resource savings compared to static hardware designs. The extent of resource savings depends on the granularity of the operations into which the application is broken down to. The evaluation also proved that partial reconfiguration based designs on reconfigurable architectures with processor cores significantly speeds-up the computations executed on the processor. The speed-up depends upon factors like number of reconfigurable containers used, the schedule of partial reconfiguration and the type of computation itself. This chapter also discussed the possible power savings on the Zynq-7000 platform by using partial reconfiguration.

Chapter 7

Conclusion and Future Work

Contents

7.1 Conclusion

This thesis presented the usage of partial reconfiguration for implementing compute intensive applications on reconfigurable architectures with a hard processor core. Most of the implemented image processing applications showed significant speed-up with the hardware accelerators implemented on the reconfigurable fabric compared to being executed completely on the processor.

Further, this work experimentally established that FPGA resource consumption reduces by time multiplexing the resources between different hardware modules using partial reconfiguration compared to a static implementation. The extent of the resource savings depend on the granularity of operations into which the application is decomposed. Finer granularity provides more resource savings. However, compared to a static implementation of an application, its implementation using partial reconfiguration is slower owing to the delay introduced by partial reconfiguration. The smaller the operations into which an application is decomposed, the higher is the number of reconfigurations required for implementing the application, which implies more overhead on account of partial reconfiguration. Based on the implementation, the reconfiguration schedule can be optimised to subsume the reconfiguration delays to achieve better performance. Thus, a design using partial reconfiguration involves trade-offs between resource consumption and performance.

The use of partial reconfiguration for implementation of applications also provides additional flexibility, for instance the implementation of morphological image operations in Chapter 5 was usable for performing the operations iteratively with minor changes to the reconfiguration schedule based in software. This is not possible in case of static hardware implementations.

To conclude, implementing applications by using partial reconfiguration provide many benefits at the cost of meagre performance penalty compared to the static hardware implementations. With optimised scheduling, it is even possible to amortise the performance penalty to a certain

extent. This design paradigm makes possible to implement certain applications which would have been otherwise difficult or even impossible.

7.2 Future Work

One of the intentions of this work was to be reusable for the OTERA[1] project for validating the developed pre-configuration and post configuration tests [14]. The synthesized hardware accelerators for the applications implemented with partial reconfiguration can be extended with the test wrappers developed for the reconfigurable fabric [15]. This way periodic online testing of the reconfigurable fabric and a configured hardware accelerator is possible.

Further, the current implementation does not use interrupts to avoid modifying the Linux kernel. The future extensions for this thesis can integrate interrupt handling mechanisms with customised interrupt handlers for the image processing applications. This way the processor need not poll on the reconfigurable containers to determine if they have finished processing the image and if the reconfiguration process is complete. After initiating the image operations and the required reconfigurations, the processor can switch to low power sleep modes or perform other software tasks. Interrupts can intimate the processor about the completion of the reconfiguration and the image operations. This could further reduce the energy required per frame during image processing.

More complex image processing pipelines can also be implemented using the designs developed as a part of this thesis. For this purpose, other Intellectual Property (IP) cores for image operations can be generated using high level synthesis.

[1] Refer Chapter 1 for details on the project

Bibliography

[1] Xilinx, *UG702: Partial Reconfiguration User Guide - v12.1*. 2010.

[2] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 165–174, 2000.

[3] S. Bhandari, S. Subbaraman, and S. Pujari, "Power Reduction in Embedded System on FPGA Using on the Fly Partial Reconfiguration," in *International Symposium on Electronic System Design (ISED)*, pp. 77–80, 2010.

[4] A. Jacobs, A. George, and G. Cieslewski, "Reconfigurable fault tolerance: A framework for environmentally adaptive fault mitigation in space," in *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 199–204, 2009.

[5] M. Shafique, L. Bauer, and J. Henkel, "Selective instruction set muting for energy-aware adaptive processors," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 353–360, 2010.

[6] L. Bauer, M. Shafique, and J. Henkel, "RISPP: A run-time adaptive reconfigurable embedded processor," in *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 725–726, 2009.

[7] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[8] S. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan, "Real Time Video Processing on FPGA Using on the Fly Partial Reconfiguration," in *International Conference on Signal Processing Systems*, pp. 244–247, 2009.

[9] D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele, "Partial reconfiguration on FPGAs in practice - Tools and applications," in *ARCS Workshops (ARCS)*, pp. 1–12, 2012.

[10] E. McDonald, "Runtime FPGA Partial Reconfiguration," in *IEEE Aerospace Conference*, pp. 1–7, 2008.

[11] C. Kohn, *XAPP1159 : Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices - v1.0*. Xilinx Inc., 2013.

[12] "OTERA: Online Test Strategies for Reliable Reconfigurable Architectures." http://www.iti.uni-stuttgart.de/abteilungen/rechnerarchitektur/projekte/otera.html, 2013. Accessed: Nov 2013.

[13] L. Bauer, C. Braun, M. Imhof, M. Kochte, H. Zhang, H. Wunderlich, and J. Henkel, "OTERA: Online test strategies for reliable reconfigurable architectures," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 38–45, 2012.

[14] L. Bauer, C. Braun, M. Imhof, M. Kochte, E. Schneider, H. Zhang, J. Henkel, and H.-J. Wunderlich, "Test Strategies for Reliable Runtime Reconfigurable Architectures," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1494–1507, 2013.

[15] J. Wang, "Online Self-Test Wrapper for Runtime-Reconfigurable Systems," Master's thesis, Institut für Technische Informatik - Universität Stuttgart, June 2013.

[16] S. Zhang, "Delay Characterization in FPGA-based Reconfigurable Systems," Master's thesis, Institut für Technische Informatik - Universität Stuttgart, December 2013.

[17] D. G. R. Bradski and A. Kaehler, *Learning OpenCV, 1st Edition.* O'Reilly Media, Inc., first ed., 2008.

[18] M. B. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays.* Springer Publishing Company, Incorporated, 1st ed., 2010.

[19] C. Kao, *Benefits of Partial Reconfiguration.* Xilinx, 2005.

[20] D. Koch, *Partial Reconfiguration on* FPGAs*: Architectures, Tools and Applications.* Springer Publishing Company, Incorporated, 2012.

[21] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, C. Braun, M. E. Imhof, H.-J. Wunderlich, and J. Henkel, "Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures," in *IEEE International Test Conference (ITC)*, pp. 1–10, 2013.

[22] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 99–104, 2009.

[23] Xilinx, *UG470: 7 Series* FPGAs *Configuration User Guide - v1.6.* 2013.

[24] H. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 205–212, 2012.

[25] D. Lim and M. Peattie, *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations.* Xilinx, Inc., 2002.

[26] K. Wu and J. Madsen, "Run-time dynamic reconfiguration: a reality check based on FPGA architectures from Xilinx," in *23rd NORCHIP Conference*, pp. 192–195, 2005.

[27] E. Eto, *XAPP290 : Difference-Based Partial Reconfiguration.* Xilinx, Inc., 2007.

[28] K. Paulsson, M. Hubner, G. Auer, M. Dreschmann, L. Chen, and J. Becker, "Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 351–356, 2007.

[29] A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong, "A novel high-performance fault-tolerant ICAP controller," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 259–263, 2012.

[30] Xilinx, *UG585 : Zynq-7000 All Programmable SoC - v1.5.* Xilinx, Inc., 2013.

[31] Altera, *Arria V Device Handbook.* 2012.

[32] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 228–235, 2011.

[33] C. Beckhoff, D. Koch, and T. Jim, "GoAhead: A Partial Reconfiguration Framework," in *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 37–44, IEEE, 2012.

[34] L. Gong and O. Diessel, "ReSim: A reusable library for RTL simulation of dynamic partial reconfiguration," in *International Conference on Field-Programmable Technology (FPT)*, pp. 1–8, 2011.

[35] F. Devic, L. Torres, J. Crenne, B. Badrignans, and P. Benoit, "SecURe DPR: Secure update preventing replay attacks for dynamic partial reconfiguration," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 57–62, 2012.

[36] K. Kepa, F. Morgan, K. Kosciuszkiewicz, and T. Surmacz, "SeReCon: A Secure Dynamic Partial Reconfiguration Controller," in *IEEE Computer Society Annual Symposium on VLSI*, pp. 292–297, 2008.

[37] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[38] G. Taubel and J.-S. Yang, "A lane departure warning system based on the integration of the optical flow and hough transform methods," in *10th IEEE International Conference on Control and Automation (ICCA)*, pp. 1352–1357, 2013.

[39] P. Daigavane and P. Bajaj, "Road Lane Detection with Improved Canny Edges Using Ant Colony Optimization," in *3rd International Conference on Emerging Trends in Engineering and Technology (ICETET)*, pp. 76–80, 2010.

[40] S. Neuendorffer, T. Li, and D. Wang, *XAPP1167 : Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries.* 2013.

[41] B. H. Fletcher, "FPGA Embedded Processors Revealing True System Performance," in *Embedded Systems Conference San Francisco*, pp. ETP–367, 2005.

[42] Xilinx, *DS190 : Zynq-7000 All Programmable SoC Overview - v1.5.* 2013.

[43] Xilinx, *UG470:* AXI *Reference Guide - v13.4.* 2012.

[44] "Zynq-7000 All Programmable SoC." http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm, 2013. Accessed: Nov 2013.

[45] J. Monson, M. Wirthlin, and B. Hutchings, "Implementing high-performance, low-power FPGA-based optical flow accelerators in C," in *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 363–369, 2013.

[46] "ZedBoard." http://www.zedboard.org/product/zedboard, 2013. Accessed: Nov 2013.

[47] Xilinx, *UG902: High-Level Synthesis - v2013.2.* 2013.

[48] Xilinx, *UG111: Embedded System Tools Reference Manual - v13.3.* 2011.

[49] "Xillinux: A Linux distribution for the Zedboard." http://xillybus.com/xillinux, 2013. Accessed: Nov 2013.

[50] A. Z. R. Langi, K. Soemintapura, T. Mengko, and W. Kinsner, "Multifractal measures of image quality," in *Proceedings of 1997 International Conference on Information, Communications and Signal Processing (ICICS)*, pp. 726–730 vol.2, 1997.

[51] Xilinx, *DS871 : LogiCORE IP Processing System 7 (v4.00.a).* 2012.

[52] Xilinx, *PG020 : LogiCORE IP AXI Video Direct Memory Access v5.02.a - Product Guide.* 2012.

[53] Xilinx, *DS768 : LogiCORE IP AXI Interconnect (v1.03.a).* 2011.

[54] Xilinx, *DS744 : LogiCORE IP AXI GPIO (v1.01.b).* 2012.

[55] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[56] Xilinx, *UG474: 7 Series* FPGAs *Configurable Logic Block User Guide - v1.5.* 2013.

[57] S. Liu, R. Pittman, A. Forin, and J.-L. Gaudiot, "On energy efficiency of reconfigurable systems with run-time partial reconfiguration," in *21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pp. 265–272, 2010.

Appendices

Appendix A

Image Processing Cores

A.1 Generation of Image Processing Cores

Image Processing Cores have been generated using the Vivado High Level Synthesis (HLS) tool and the integrated HLS Video Library which provides synthesizable version of OpenCV primitives. Vivado HLS tool is a high level synthesis tool used for generating Register Transfer Level (RTL) descriptions of programs written in C/C++. Generally C/C++ programs follow sequential behaviour while the corresponding hardware is generally concurrent. In order to generate hardware description of a program, this tool converts the sequential behaviour of the program into parallel behaviour of hardware.

For this purpose the tool must be informed about the possibilities of parallelism, pipelining etc. in the program. All the information is provided to the tool in the form of compiler directives. To begin with the synthesis, the project setting of the tool is updated with the name of the top-level C/C++ function to be synthesized and the desired clock speed with which the synthesized modules are to be used. This function is synthesized as a hardware module with all the function arguments as ports. The tool may add additional ports for the interface and output signals, especially if the function returns any value. The program being synthesized can use compiler directives to advise the tool to wrap the ports into bus interfaces. Vivado HLS supports AXI4-Lite Slave, AXI4-Master and AXI4-Streams for this purpose.

This thesis generates image processing IP cores wrapped in different AXI4 bus interfaces with constraints of being clocked with 100 MHz.

A.2 Resource Usage of Image Processing Cores

This Table A.1 shows the resource usage of all the image processing cores generated from the Vivado High Level Synthesis tool and the HLS Video Library. The reconfigurable resources available in the Zynq-7000 device used in this thesis are described in Chapter 6.

The entry with containers presents the maximum resources available within the reconfigurable container to accommodate the image processing IP cores. All the resource consumption shown in the table are pre-placement estimations. The design flow recommends using reconfigurable

	Resource Consumption for 640 x 480 image processing					
	Registers	**Flip-Flops**	**SliceL**	**SliceM**	**DSP**	**RAM(18Kb)**
Containers	4800	9600	700	500	40	20
Loopback	1010	586	131	122	0	0
Loopback 2	1177	698	148	147	0	0
Sobel	2872	1731	408	311	0	5
Dilation	1783	1529	241	206	0	5
Erosion	2097	1618	288	237	0	5
Threshold	2035	1060	267	242	0	0
Convert to Gray	1155	673	147	142	3	0
Opening	3406	2647	484	368	0	9
Closing	3406	2647	484	368	0	9
Noise Filter	5028	2978	734	523	21	8
Morp. Gradient	3590	2738	508	390	0	9
Image Subtract	1328	865	166	167	0	0

Table A.1: Resource consumption of all generated IP cores

modules smaller than the reconfigurable containers. However, the constraints are not stringent as seen in case of Noise Filter. Despite having estimations indicating over-usage of registers, the tool chain could successfully place and route it in all the suitable reconfigurable containers.

	Bitstream sizes in KB		
	Container 1	**Container 2**	**Container 3**
Bitstream w/o compression	317.4	317.4	317.4
Loopback	211.1	213.3	202.3
Sobel	218.5	222.9	219.3
Convert To Gray	204.6	217.4	206.7
Erosion	218.7	218.5	219.3
Dilation	222.9	218.5	219.3
Opening	222.9	222.9	223.3
Closing	222.9	223.2	223.2
Threshold	214.4	214.2	215.1
Morphological Gradient	222.9	222.9	223.3
Noise Filter	231.2	231.1	231.1

Table A.2: Size of Partial Bitstreams

The Table A.2 shows the size of all the generated partial bitstreams for the design using independent containers. The uncompressed size of all partial bitstreams meant for a particular reconfigurable container is always same. However their size can be reduced by compression and the compressed size can be treated as an indication of resource usage. For generating compressed bitstreams, the bit generator invoked from the PlanAhead tool must set appropriate

options.

As seen from Table A.1, Loopback IP core had the least resource consumption while Noise Filter had the highest. This is apparent from the size of their partial bitstreams as well.

A.3 Performance of Image Processing Cores

The Table A.3 compares the performance of image processing operations in software and using the IP cores generated using the Vivado tool flow. It is important to note that the timings for the execution of image processing operations using the IP core can only be taken as an indication due to the inaccuracies observed in the measurement on account of context switching as explained in Chapter 6.

	Processing time (in ms) for 640 x 480 image	
	Software (OpenCV)	**Hardware (IP Cores)**
Loopback	0	3.7
Erosion	45.1	3.25
Dilation	45.3	3.4
Opening	90.3	3.45
Closing	90.6	3.35
Convert to Gray	8.2	4
Threshold	2.55	3.6
Sobel	132.8	4.1
Noise Filter	45.4	3.55
Morp. Gradient	104.6	3.4
Image Subtraction	4.3	4.2

Table A.3: Performance of Image Processing Cores

As shown in the Table A.3, the hardware execution gives faster results as compared to the software implementations for compute intensive image processing operations. For simpler operations like subtraction and thresholding, the performance of hardware is comparable with software.

A.4 Power Consumption of Image Processing Core

The power consumption of the generated image processing IP cores is presented in the Table A.4. The table shows static and dynamic power consumption for each of the generated IP core. The power consumption is measured by the use of the Xilinx XPower Analyzer tool. This tool takes as input the implemented design and provides an estimate of static and dynamic power consumption for the hierarchical design using the provided test bench. In absence of a

test bench, the tool assumes switching probabilities for all signals in the design for estimating the power.

	Power Consumption of Image Processing IP cores		
	Total Power	**Static Power**	**Dynamic Power**
Loopback	0.35	0.12	0.23
Sobel	0.45	0.16	0.29
Convert to Gray	0.39	0.14	0.25
Erosion	1.23	0.92	0.31
Dilation	1.24	0.95	0.29
Opening	0.99	0.6	0.39
Closing	0.99	0.6	0.39
Threshold	0.7	0.2	0.5
Morphological Threshold	2.23	1.76	0.47
Noise Filter	0.53	0.2	0.33

Table A.4: Power Consumption of the Image Processing IP Cores (in mW)

The power consumptions for each of the generated IP cores have been measured without developing specific test benches. The tool hence estimated the power consumption based on the default probabilities and hence the figures may not be very accurate.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature