Dynamic HW/SW Partitioning: Configuration Scheduling and Design Space Exploration

by

Santheeban Kandasamy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Santheeban Kandasamy

Abstract

Hardware/software partitioning is a process that occurs frequently in embedded system design. It is the procedure of determining whether a part of a system should be implemented in software or hardware. This dissertation is a study of hardware/software partitioning and the use of scheduling algorithms to improve the performance of dynamically reconfigurable computing devices. Reconfigurable computing devices are devices that are adaptable at the logic level to solve specific problems [Tes05]. One example of a reconfigurable computing device is the field programmable gate array (FPGA). The emergence of dynamically reconfigurable FPGAs made it possible to configure FPGAs at runtime. Most current approaches use a simple on demand configuration scheduling algorithm for the FPGA configurations. The on demand configuration scheduling algorithm reconfigures the FPGA at runtime, whenever a configuration is needed and is found not to be configured. The problem with this approach of dynamic reconfiguration is the reconfiguration time overhead, which is the time it takes to reconfigure the FPGA with a new configuration at runtime. Configuration caches and partial configuration have been proposed as possible solutions to this problem, but these techniques suffer from various limitations.

The emergence of dynamically reconfigurable FPGAs also made it possible to perform dynamic hardware/software partitioning (DHSP), which is the procedure of determining at runtime whether a computation should be performed using its software or hardware implementation. The drawback of performing DHSP using configurations that are generated at runtime is that the profiling and the dynamic generation of configurations require profiling tool and synthesis tool access at runtime. This study proposes that configuration scheduling algorithms, which perform DHSP using statically generated configurations, can be developed to combine the advantages and reduce the major disadvantages of current approaches. A case study is used to compare and evaluate the tradeoffs between the currently existing approach for dynamic reconfiguration and the DHSP configuration scheduling algorithm based approach proposed in the study. A simulation model is developed to examine the performance of the various configuration scheduling algorithms. First, the difference in the execution time between the different approaches is analyzed. Afterwards, other important design criteria such as power consumption, energy consumption, area requirements and unit cost are analyzed and estimated. Also, business and marketing considerations such as time to market and development cost are considered.

The study illustrates how different types of DHSP configuration scheduling algorithms can be implemented and how their performance can be evaluated using a variety of software applications. It is also shown how to evaluate when which of the approaches would be more advantageous by determining the tradeoffs that exist between them. Also the underlying factors that affect when which design alternative is more advantageous are determined and analyzed. The study shows that configuration scheduling algorithms, which perform DHSP using statically generated configurations, can be developed to combine the advantages and reduce some major disadvantages of current approaches. It is shown that there are situations where DHSP configuration scheduling algorithms can be more advantageous than the other approaches.

Acknowledgements

First, I would like to thank my family – Mom, Dad, Brothers, Sisters, Uncles, Aunts and Diana. I could not have completed this without your support.

I would like to thank my supervisors Dr. Andrew Morton and Dr. Wayne Loucks. Thank you for making this experience into an interesting and enjoyable one. I am very glad that I had the opportunity to work with you. Thank you for your guidance, support and always being around when I needed help and advice.

Thanks to CMC Microsystems for providing the lab equipment and design tools. I am also thankful to Dr. Russell Tessier, Dr. William Bishop, Dr. Frank Vahid and Greg Stitt for their advice and guidance. Finally I would like to thank my friends for all the entertainment and sport events.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

This dissertation is a study of hardware/software partitioning and the use of scheduling algorithms to improve the performance of dynamically reconfigurable computing devices. Hardware/software partitioning is the procedure of determining whether subsystem should be implemented in software or hardware and it is one of the design stages of the hardware/software co-design flow. Hardware/software co-design is the simultaneous design of hardware and software to obtain an optimized system. It is frequently used for the design of embedded systems since it exploits the synergy of considering hardware and software design issues concurrently and therefore allows the design of highly optimized systems.

Reconfigurable computing devices are hardware devices that are adaptable at the logic level to solve specific problems [Tes05]. One example of a reconfigurable computing device is the field programmable gate array (FPGA), which is in wide use today for system customization, glue logic and rapid prototyping. FPGAs are configurable integrated circuits that have programmable interconnections and logic. They have the advantage of being very flexible and they do not have to be custom manufactured since they provide post fabrication customization.

1.1 Dynamic Reconfiguration

The emergence of dynamically reconfigurable FPGAs made it possible to configure FPGAs at runtime. Dynamic reconfiguration is useful in systems where only a subset of the configurations is needed at a time. One advantage of dynamic reconfiguration is that a single smaller FPGA can be time-multiplexed to support multiple configurations over time. In contrast, a statically configured system would require multiple FPGAs or a larger FPGA to hold all the configurations at all times. Dynamically reconfigurable systems therefore have a higher functional density due to the use of the smaller FPGA, which can result in reduced unit cost and area requirements. The configuration scheduling algorithm (CSA) is the algorithm that determines when a configuration is used for device reconfiguration. Most approaches use a simple on demand (OD) configuration scheduling algorithm. The on demand configuration scheduling algorithm configures the reconfigurable computing device at runtime, whenever a new configuration is needed. The problem with this approach of dynamic reconfiguration is the reconfiguration time overhead, which is the time it takes to configure the FPGA with a new configuration at runtime.

To address the reconfiguration time overhead, configuration caches and dynamically programmable gate arrays (DPGAs) have been proposed in [BD94], [HW97] and [JC99]. Frequently used configurations can be kept in a configuration cache, which allows for fast reconfiguration and therefore reduces the reconfiguration time overhead. DPGAs are FPGAs with multiple contexts. These allow for fast switching between configurations. But up to now there has still not been any commercial implementation of a DPGA. This is due to the fact that the extra area and cost associated with the configuration caches and multiple contexts of DPGAs acts against one of the main objectives of dynamic reconfiguration, which is the goal of obtaining a higher functional density from a smaller FPGA. Another approach to reduce the reconfiguration time overhead is to use partial reconfiguration. Partial reconfiguration involves the reconfiguration of a portion of the FPGA while leaving the previous configuration for the rest of the FPGA. By only having to reconfigure parts of the FPGA the reconfiguration time is reduced. To take advantage of this approach it is required that the hardware designs to be configured in the FPGA have parts in common. Advanced electronic design automation (EDA) tools are often required and used in the industry to identify such common parts of the hardware designs, which results in additional development costs and an increase in time to market (TTM).

The emergence of dynamically reconfigurable FPGAs also made it possible to perform dynamic hardware/software partitioning (DHSP), which is the procedure of determining at runtime whether a computation should be performed using its software or hardware implementation. Modern EDA tools have the potential of fully automating the process of hardware/software partitioning and enable the possibility of moving hardware/software partitioning and the generation of the FPGA configurations from the design phase to the execution stages of the system. Such an approach was studied in [SL03] and [LV04] where the software is initially executed without any hardware acceleration, while the software is being profiled at runtime. After determining the portion of the software that would benefit most from hardware acceleration a coprocessor is generated for that part of the software at runtime. Since the decision of which part of the software should be executed using its software or hardware implementation is made at runtime, this approach is called dynamic hardware/software partitioning (DHSP). The drawback of performing DHSP using configurations that are generated at runtime is that the profiling and the dynamic generation of configurations require profiling tool access and synthesis tool access at runtime.

1.2 Thesis Statement

Many past research projects showed the advantages of DHSP and dynamic reconfiguration of FPGAs in general. But there have still not been any significantly successful applications of these technologies in industry, due to various weaknesses and limitations associated with them. Most current approaches for dynamic reconfiguration of FPGAs use a simple on demand configuration scheduling algorithm that reconfigures the FPGA at runtime, whenever a new configuration is needed. The problem with this approach of dynamic reconfiguration is the reconfiguration time overhead. The approach of performing DHSP using dynamically generated configurations has the drawback that the profiling and the dynamic generation of configurations require profiling tool access and synthesis tool access at runtime. It is my thesis that configuration scheduling algorithms, which perform DHSP using statically generated configurations, can be developed to combine the advantages and eliminate the major drawbacks of both the on demand configuration scheduling algorithm and DHSP using dynamically generated configurations. This new approach eliminates the need for profiling and synthesis tool access at runtime, by using statically generated configurations. By performing DHSP, the configuration scheduling algorithm can also reduce the reconfiguration time overhead.

1.3 Thesis Contribution

The main contributions of the study are:

1. illustration of how configuration scheduling algorithms can be used to perform DHSP using statically generated configurations,
2. implementation of different types of DHSP configuration scheduling algorithms and evaluation of how they perform on a variety of software applications,
3. comparison and evaluation of the tradeoffs between the current approach for dynamic reconfiguration (on demand configuration scheduling algorithm) and the DHSP configuration scheduling algorithm based approach proposed in the study, and
4. analysis and determination of the underlying factors that affect when a design alternative is more advantageous than another.

In Chapter 6 these contributions are evaluated and the conclusions of the study are summarized.

1.4 Thesis Outline

Chapter 2 introduces the research area of the study by reviewing the existing literature. An introduction to hardware/software co-design and reconfigurable computing is presented first, followed by a discussion on dynamic reconfiguration and hardware/software partitioning. Afterwards some of the current challenges that exist in these research areas are identified. Then the methods that are currently proposed in the literature to solve the problems examined in the study are analyzed and appraised. Chapter 3 first looks at the effects of hardware/software partitioning on the execution time of a software application. Then it is identified how the on demand configuration scheduling algorithm affects the application's execution time. Afterwards, the approach of performing DHSP using configuration scheduling algorithms is proposed. Some benefits of the DHSP configuration scheduling algorithm based approach and its impact on the application's execution time are illustrated and analyzed

Chapter 4 first presents a case study that is used to evaluate the different approaches and configuration scheduling algorithms. Afterwards the modeling framework and methodology that was chosen to examine the performance of the configuration scheduling algorithms is explained. After introducing the implementation of the model, simulation results are presented. The results are then examined and it is determined if they are consistent with the analysis and theory presented in Chapter 3. Finally the patterns and trends in the simulation results are identified and the insights gained from the experiments are summarized. Chapter 4 examines the difference in the execution time between the design alternatives. Chapter 5 first examines and estimates other important design criteria such as power consumption, energy consumption, area requirements and unit cost. Afterwards business and marketing considerations such as TTM and development cost are also considered. The goal of this chapter is to provide insight into how to evaluate when a design alternative is advantageous by determining the tradeoffs that exist in the design space. Chapter 6 first summarizes the contributions and observations made in the study. Afterwards suggestions on the direction of future research in the area of the study are provided.

Chapter 2

Background and Literature Review

In this chapter, the research area of the study is introduced by reviewing the existing literature. First a brief introduction to hardware/software co-design and reconfigurable computing is presented, followed by a discussion on dynamic reconfiguration and hardware/software partitioning. After identifying some of the current challenges that exist in these research areas, methods, which are currently proposed in the literature to solve the problems examined in the study, are analyzed and appraised.

2.1 Hardware/Software Co-Design

Hardware/software co-design is the process of designing hardware and software in parallel to obtain an optimized system. This section provides a brief introduction to the hardware/software co-design flow and illustrates where hardware/software partitioning falls in the design flow.

2.1.1 Design Flow

Co-design is frequently used for the design of embedded systems, since they require optimization to satisfy various design constraints and objectives. Modern advances in electronic design automation (EDA) tools such as high level synthesis tools and simulation tools make it possible to explore both hardware and software design alternatives in parallel, to select the best alternative. One advantage of designing an embedded system using co-design is that it reduces the time to market (TTM), by allowing the development of hardware and software in parallel. Hardware/software co-design also exploits the synergy of considering hardware and software design issues concurrently and therefore allows the design of highly optimized systems.

It is possible to design both hardware and software in parallel when designing embedded systems, since the workload, which is the work that needs to be performed by the system, is well defined at hardware design time. This is in sharp contrast to the way general purpose systems, such as personal computers (PCs) are designed. For such systems, the exact workload is not known at hardware design time and the hardware is designed well in advance before the software. The entire hardware/software co-design flow is illustrated in Figure 2-1.

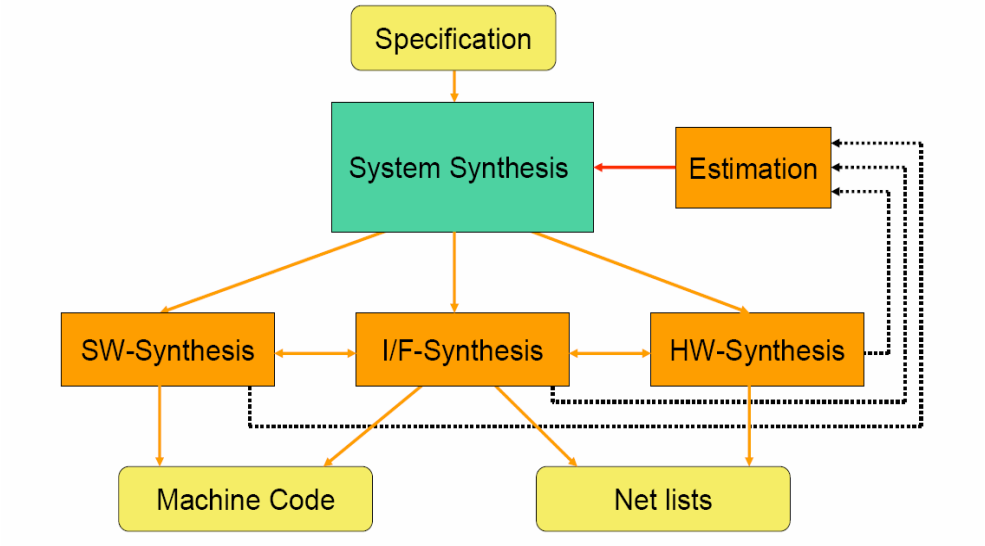

Figure 2-1: Hardware/Software Co-Design Flow [Thi06]

Synthesis is the process of transforming a behavioral description of a system into a structural description by moving the description to a lower abstraction level. The system synthesis phase takes the specification, which describes the desired functionality of the system and determines a possible architecture of the system. This architecture is composed of the structural descriptions of the hardware, software and the interface between them. The software architecture is described using modules and their interaction and the hardware architecture is described using components and their interconnections. It is then possible to take this high level description of the system and synthesize the hardware and software to lower abstraction levels, such as machine code and net lists, using compilers and hardware synthesis tools. The synthesis output can then be used to estimate the values of important design criteria, which would be obtained if this particular design alternative was chosen. By iterating through various design alternatives the design space is explored and values of design criteria, which are of interest, are estimated. Finally the estimated values of the design criteria are used to select the best design alternative. This process of analyzing several functionally equivalent design alternatives to determine the most suitable one is known as design space exploration. For the purposes of the study the next section takes a closer look at the system synthesis phase and hardware/software partitioning in particular.

2.1.2 System Synthesis

To show where hardware/software partitioning falls in the co-design flow the details of the system synthesis phase are presented in Figure 2-2 and each of the system synthesis stages are described thereafter.

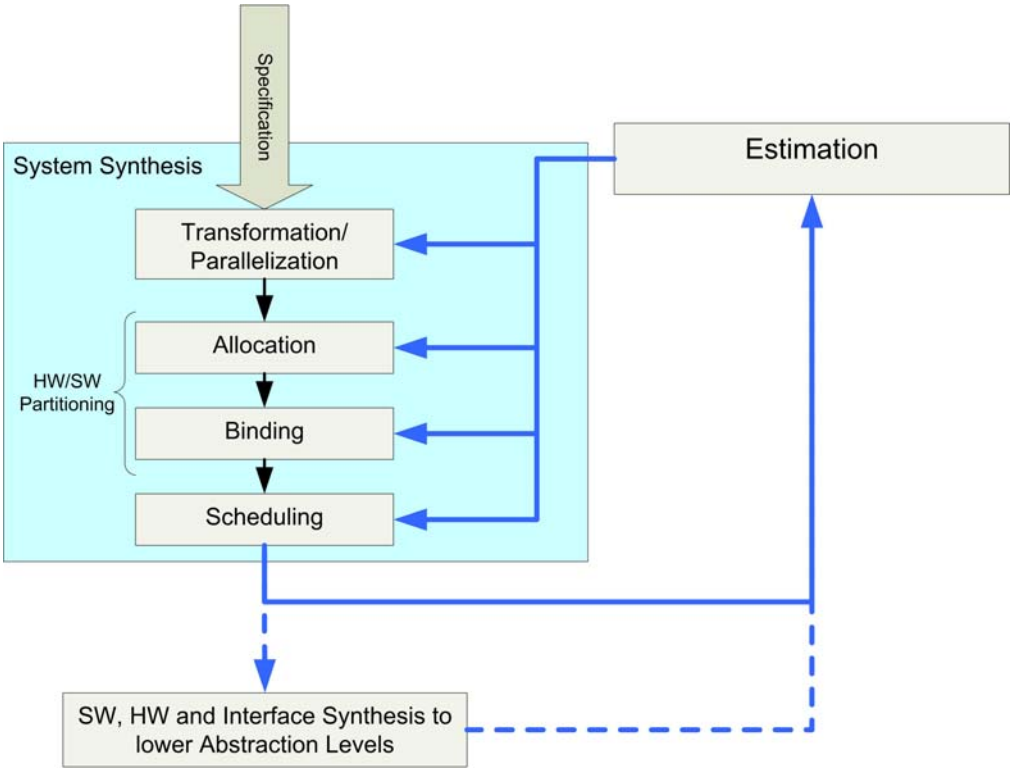

Figure 2-2: System Synthesis

Transformation/Parallelization: During this stage complex systems are split up into multiple simpler processes that can be executed individually to satisfy the requirements of the entire system. At this design stage it is still unclear how the processes are implemented in the final system. These considerations are left to the later stages in the design flow. At this point, the only concern is about how to split the work up into smaller pieces using a divide and conquer strategy. The goal of parallelization is to improve performance by allowing work to be done concurrently but it also splits up the system into smaller more manageable pieces.

Allocation: Allocation is the process of selecting the hardware architecture that will be used for implementing the system. This involves selecting the number and types of processors that will be used. Selecting the memory architecture and interconnections between the various components is also part of this step. From the transformation/parallelization stage, a set of processes and the communication and synchronization that need to be present between the processes are defined. Now processors need to be selected to implement the processes. For the purposes of the study, processors can be categorized using the following classifications.

- Processor Technologies
    1. General Purpose Processor (GPP) – software implementation
    2. Single Purpose Processor (SPP) – hardware implementation using a coprocessor
- Integrated Circuit (IC) Technologies
    1. Full Custom ASIC
    2. Gate Array based ASIC
    3. Field Programmable Gate Array (FPGA)

The main advantage of GPPs is that they are flexible since they can perform any type of computation. SPPs are not flexible and suffer from high development costs, but perform the computation that they are customized to execute at a higher speed than GPPs. Full custom ASIC technology has a high development cost due to the custom placement and routing that is required. Gate array based ASICs have pre-placed arrays of gates and therefore the only customization that is required is the routing between the gates. Gate array based ASICs are less optimal compared to full custom ASICs in terms of performance and power but the development cost of gate array based ASICs is significantly less. FPGAs are configurable ICs that have programmable interconnections and logic. This type of IC is the most flexible; it requires the least amount of hardware development cost and provides fast TTM, but it also provides the worst performance, power, and area requirements when compared to the other IC technologies. Any processor technology can be mapped to any type of IC technology.

Binding: A set of processes and the communication and synchronization that need to be present between the processes are now defined. Also the hardware architecture, which consists of a set of processors, other hardware components and their interconnections, is now defined. In the binding stage the processes are mapped to processors and the data and instructions are partitioned to memory.

Scheduling: Having the processes assigned to specific processors, it is necessary to schedule when each process executes on the processors. Scheduling the processes on the processors is the goal of this stage.

After introducing the system synthesis stages it can now be examined where hardware/software partitioning falls in the co-design flow. Hardware/software partitioning is the procedure of determining whether a part of a system should be implemented on a GPP (software) or SPP (hardware). Hardware/software partitioning can be thought of as taking place over the allocation and binding stages of system synthesis as shown in Figure 2-2. Since the hardware/software partitioning problem occurs frequently in embedded system design much research has been devoted to analyze this problem. Since hardware/software partitioning will be the main focus of the study, this section was dedicated to introduce where it falls into the broader research area of hardware/software co-design.

2.2 Reconfigurable Computing

This section provides a brief introduction to reconfigurable computing followed by a discussion on dynamic reconfiguration and dynamic hardware/software partitioning (DHSP). Some of the current challenges that exist in these research areas are also presented. Also the techniques, which are currently proposed in the literature to solve the limitations of existing methods, are appraised.

Reconfigurable computing devices are hardware devices that are adaptable at the logic level to solve specific problems [Tes05]. One example of a reconfigurable computing device is the previously introduced FPGA, which is in wide use today for system customization, glue logic and IC prototyping. FPGAs are configurable ICs that have programmable interconnections and logic. They have the advantage of being very flexible and they do not have to be custom manufactured since they provide post fabrication customization. Due to this property, FPGAs require the least amount of hardware development cost and provide fast TTM. Due to the overhead of being reconfigurable, they provide the worst performance, power and area requirements when compared to full custom and application specific ICs. For the purposes of the study, FPGAs and reconfigurable computing devices in general can be categorized using the following classifications:

- FPGA coupling to Central Processing Unit (CPU)
    1. FPGA/coprocessor connected to CPU through a bus – loose coupling
    2. CPU with a configurable functional unit (FU) – tight coupling
- Type of configuration
    1. Static configuration –FPGA is configured once before use and not at runtime
    2. Dynamic configuration – FPGA is reconfigured multiple times at runtime

2.2.1 Dynamic Reconfiguration

One advantage of dynamic reconfiguration is that a single smaller FPGA can be time multiplexed to support multiple configurations over time. Dynamic reconfiguration is useful in systems where only one of the configurations is needed at a time. In contrast, a statically configured system would require multiple FPGAs or a larger FPGA to hold all the configurations at all times. Dynamically reconfigurable systems therefore have a higher functional density due to the smaller FPGA, which results in reduced unit cost and area.

One application of dynamic reconfiguration is to update systems and to fix problems with previous releases. It is possible, for example, to update the FPGA with a new configuration at runtime to support a newer communication protocol. Dynamic reconfiguration can also be used to increase the fault tolerance of a FPGA design. Once faulty locations have been detected in the FPGA, it is possible to reconfigure the device with an alternate and functionally equivalent configuration of the same design, such that the faulty locations of the FPGA stay unused. The alternate configurations can be recompiled and generated at runtime based on fault location information or multiple configurations can be precompiled and stored in anticipation of faults. Generating alternate configurations at runtime requires access to synthesis tools and suffers from long system downtime due to the time it requires to generate the new configurations. Storing precompiled configurations of the same design suffers from additional memory overhead, but provides significantly faster reconfiguration and eliminates the need for synthesis tool access at runtime. In [HM01], multiple precompiled configurations of the same design with different unused columns of the FPGA are generated during the design phase. When a fault is discovered the FPGA is reconfigured with one of the precompiled configurations, which does not use the faulty column.

As mentioned previously, dynamic reconfiguration is also useful in applications with time-varying behavior. The FPGA can be reconfigured dynamically with an optimal configuration for a given time, in

response to changing environmental inputs to the system. In [ST02], this concept is illustrated using a dynamically reconfigurable adaptive Viterbi decoder that is implemented on a FPGA. The amount of computation, which needs to be performed to decode the transmitted data with an acceptable bit error rate, is proportional to the channel noise. The adaptive Viterbi decoder takes advantage of the fact that the channel noise varies over time. The decoder is configured such that it performs the minimum amount of computation needed at a given time depending on the amount of noise present in the channel. When an increase or decrease in the amount of channel noise is detected, the decoder is dynamically reconfigured to perform more or less computation respectively. This allows the decoder to adapt to the varying channel noise, instead of always assuming the worst case channel noise. Dynamic reconfiguration therefore allows the decoder to tune and adapt to varying channel conditions, which results in more efficient computation and the maximum performance possible at a given time.

Dynamic reconfiguration can be performed with both tightly and loosely coupled reconfigurable computing devices. The PRISC and Chimaera projects [Tes05] are examples of tightly-coupled reconfigurable functional units. These functional units are often used to augment the existing instruction set. They share the processor registers and depend on the processor for access to memory. Loosely-coupled reconfigurable computing devices on the other hand are independent coprocessors that have their own interface for memory access. Garp [HW97] is an example of a loosely-coupled system designed for streaming data applications. It consists of a MIPS processor with a reconfigurable coprocessor. Here the software executing on the MIPS explicitly reconfigures the coprocessor using precompiled configurations whenever a needed configuration is not present in the coprocessor. One-Chip [JC99] is a processor with a reconfigurable functional unit that has direct access to memory. It is therefore considered to be a hybrid between a reconfigurable functional unit and coprocessor. Here the software must first initialize a reconfiguration bits table, with memory addresses that indicate where the different configurations can be found in memory. Whenever a needed configuration is not currently configured, the hardware determines the address of the configuration data from the reconfiguration bits table and performs the reconfiguration.

The configuration scheduling algorithm (CSA) is the algorithm that determines when a configuration is used for device reconfiguration. The reconfigurable computing devices introduced so far use a simple on demand (OD) configuration scheduling algorithm. The on demand configuration scheduling algorithm reconfigures the reconfigurable computing device at runtime, whenever a new configuration is needed. A detailed discussion on configuration scheduling algorithms is the topic of Chapter 3.

One common problem with the all of the techniques examined so far is the time it takes to reconfigure the device with a new configuration as identified in [HW97], [Tes05] and [JC99]. This reconfiguration time overhead becomes a significant problem when the frequency of reconfiguration (FRC) increases. As the frequency of reconfiguration increases, the time intervals between device reconfiguration decreases, which magnifies the performance degradation due to the reconfiguration time overhead. It is possible to decrease the negative effects of the reconfiguration time overhead by decreasing the frequency of reconfiguration as shown in [ST02]. Reducing the frequency of reconfiguration can also have other positive benefits such as reducing the power consumption of the system. It should also be noted that reducing the frequency of reconfiguration is not a possibility in many systems due to the inherent inflexibility of the configuration scheduling algorithm employed. When using an on demand configuration scheduling algorithm, there is no other possibility than reconfiguring the device with a required configuration, when it is needed and is found not to be currently configured.

To address the reconfiguration time overhead, configuration caches and dynamically programmable gate arrays (DPGAs) have been proposed in [BD94], [HW97] and [JC99]. Frequently used configurations can be kept in a configuration cache, which allows for fast reconfiguration and therefore reduces the reconfiguration time overhead. DPGAs are FPGAs with multiple contexts, which allows for fast switching between configurations. But up to now there has still not been any real implementation of a DPGA. This is especially due to the fact that one could simply use a larger FPGA capable of holding all the configurations statically, instead of dealing with the problems associated with DPGAs, configuration caches and dynamic reconfiguration in general. The extra area and cost associated with the configuration caches and multiple contexts of DPGAs acts against one of the main objectives of dynamic reconfiguration, which is the goal of obtaining a higher functional density from a smaller FPGA.

Another approach to reduce the reconfiguration time overhead is to use partial reconfiguration. Partial reconfiguration involves the reconfiguration of parts of the FPGA while leaving the previous configuration for the rest of the FPGA. By only having to reconfigure parts of the FPGA the reconfiguration time is reduced. Some commercial FPGAs such as the Xilinx Virtex II support partial reconfiguration [Xil06a]. To take advantage of this approach it is required that the hardware designs to be configured in the FPGA have parts in common. Advanced EDA tools are often required to identify such common parts of the hardware designs, which results in additional development costs and an increase in the TTM.

2.2.2 Dynamic Hardware/Software Partitioning

Most of the reconfigurable computing devices introduced so far use precompiled configurations, which are statically generated at design time and then used for dynamic reconfiguration at runtime. One exception to this is the generation of alternate configurations at runtime, to increase the fault tolerance of the FPGA as discussed in [HM01]. Here alternate configurations are generated dynamically based on fault location information, such that faulty locations of the FPGA stay unused. This approach suffers from the need for synthesis tool access at runtime. Dynamic reconfiguration can therefore be categorized into static and dynamic generation of configurations.

As pointed out previously hardware/software partitioning is the procedure of determining whether a part of a system should be implemented on a GPP (software) or SPP (hardware). The emergence of dynamically reconfigurable devices such as FPGAs made it possible to perform DHSP, which is the procedure of determining at runtime whether a computation should be performed on a GPP or SPP. Hardware/software partitioning can then be categorized using the following classifications.

- Hardware/Software Partitioning

    1. Static – Partitioning performed at design time
    2. Dynamic – Partitioning performed at runtime

Modern EDA tools have the potential of almost fully automating the process of static hardware/software partitioning. [HT04a], [HT04b] and [Cri05] introduce Critical Blue's automated tools, which take compiled software binary code as input and produce a coprocessor that accelerates the critical kernels of the software. The tool flow consists of first determining the critical kernels of the software using a profiler. Afterwards the synthesis tools automatically generate a coprocessor that accelerates the critical kernels. The original software code is also automatically modified to perform communication with the coprocessor whenever the critical kernel needs to be executed. These tools therefore show that it is possible to fully automate hardware/software partitioning and the synthesis of coprocessors using only compiled binary code of the software application as input. Advanced EDA tools such as these and the emergence of dynamically reconfigurable computing devices enabled the possibility of moving hardware/software partitioning and the generation of the coprocessors from the design phase to the execution stages of the system.

Such an approach was studied in [SL03] and [LV04]. Here the system architecture consists of a GPP, memory, a dynamic partitioning module and a reconfigurable logic module. Initially the GPP executes the software without any hardware acceleration, while the dynamic partitioning module profiles the software by monitoring the bus accesses between the GPP and the memory. After determining a suitable critical kernel the dynamic partitioning module generates a coprocessor for the critical kernel and configures the reconfigurable logic module with the coprocessor. The software executing on the GPP is also modified to communicate with the coprocessor for future executions of the critical kernel. This approach also allows for the detection of changes in the behavior of the software due to external inputs. Once a different critical kernel is found to be executing more frequently than the one accelerated through the current coprocessor, the reconfigurable logic module can be reconfigured with a new coprocessor to accelerate the more frequent critical kernel. Since the decision of whether a critical kernel should be executed using the GPP or a coprocessor is made at runtime, his approach is called dynamic hardware/software partitioning (DHSP).

One advantage of this approach is that the hardware acceleration of the software is performed seamlessly to the software developer. The software developer does not need to perform any profiling or synthesis of coprocessors. There is also no need to make any platform specific code changes, which preserves the portability of the software. Another advantage of this approach is that it reduces the frequency of reconfiguration and eliminates the reconfiguration time overhead. Always having the ability to execute the critical kernel using the GPP allows for the reconfiguration of the reconfigurable logic module to be performed concurrently, while the GPP executes the software without any hardware acceleration. Due to this, the reconfiguration time becomes a less crucial factor to the performance of the system. Some of the drawbacks of this approach are due to fact that the detection of critical kernels and the dynamic generation of coprocessors require profiling tool access and synthesis tool access at runtime. The approach in [SL03] and [LV04] attempts to solve this problem by using simplified synthesis tools and by using special reconfigurable logic architectures. The simplified synthesis tools were only able to generate coprocessors with purely combinational circuits. Also the extra area, cost and power consumption associated with the dynamic partitioning module outweighs the possible benefits of such an approach. A similar study with similar results and conclusions to [SL03] and [LV04] using a tightly coupled reconfigurable functional unit instead of a loosely-coupled coprocessor was performed in [BC05].

2.3 Summary

This chapter introduced the research area of the study by reviewing the existing literature. First hardware/software co-design was presented as the process of designing hardware and software in parallel to obtain an optimized system. Then it was explained where hardware/software partitioning falls in the design flow and it was explained that hardware/software partitioning is the procedure of determining whether a part of a system should be implemented on a GPP or SPP.

Then an introduction to reconfigurable computing and dynamic reconfiguration was provided. Dynamic reconfiguration is useful in systems where only one configuration is needed at a time. One advantage of dynamically reconfigurable systems is the higher functional density due to the smaller FPGA. Afterwards some applications of dynamic reconfiguration such as runtime system updates, increasing fault tolerance and adaptive decoding were introduced. Dynamic reconfiguration was categorized into systems using statically or dynamically generated configurations. One common problem of dynamic reconfiguration using statically generated configurations is the reconfiguration time overhead. This overhead becomes a significant problem when the frequency of reconfiguration increases. Configuration caches and DPGAs are approaches to solving this problem, but the extra area and cost associated with them outweigh their benefits. Partial reconfiguration can only be used to address the reconfiguration time problem in situations when parts of the hardware designs to be configured in the FPGA are in common.

DHSP was then introduced and an approach using dynamic configuration generation was examined. One advantage of this approach is that it reduces the frequency of reconfiguration and eliminates the reconfiguration time overhead. One of the drawbacks of this approach is that it requires profiling and synthesis tool access at runtime. The extra area, cost and power consumption associated with the DHSP using dynamic configuration generation outweighs the possible benefits of such an approach.

Even though there have been a lot of research projects, which show the advantages of dynamic reconfiguration and DHSP, there have still not been any significantly successful applications of these technologies used in the industry, due to various weaknesses and limitations that were pointed out. Another weakness of these technologies, which was not mentioned earlier, is the lack of high-level simulation tools to evaluate the advantages and disadvantages of using dynamic reconfiguration and DHSP as compared to using static reconfiguration and static hardware/software partitioning respectively.

Currently most managers and system architects avoid using dynamic reconfiguration and DHSP, due to the difficulty of evaluating these techniques as possible design alternatives at the conceptual design phase.

The rest of the study addresses the reconfiguration time overhead and the high frequency of reconfiguration, which are some of the weaknesses and limitations of dynamic reconfiguration and DHSP that were identified in this chapter. The algorithm that determines when a configuration is used for device reconfiguration was defined as the configuration scheduling algorithm. It was pointed out that a simple on demand (OD) configuration scheduling algorithm is currently applied in most systems that use statically generated configurations. In the next chapter an approach for performing DHSP using a new type of configuration scheduling algorithm is proposed.

Chapter 3

Execution Time and Configuration Scheduling

This chapter first looks at the effects of hardware/software partitioning on the execution time of a software application. Then, it is identified how on demand configuration scheduling algorithms, which are used in many dynamically reconfigurable systems, affect the application's execution time. Later on, an approach for performing DHSP using a new type of configuration scheduling algorithm is proposed. After illustrating some benefits of the new configuration scheduling algorithm and analyzing its impact on the application's execution time, it is explained how the rest of the study was approached.

3.1 Hardware/Software Partitioning

This section examines hardware/software partitioning and its impact on the execution time of a software application. As introduced previously, hardware/software partitioning is the procedure of determining whether a part of a system should be implemented on a GPP or SPP. The SPP could be implemented on any IC technology. In the study, a specific type of hardware/software partitioning problem, which arises frequently in embedded system design, is considered. A software application is considered, for which a software profiler is used to identify its critical kernels. Critical kernels are loops in the software code that consume a significant amount of computation time. On average, software applications spend most of their time executing critical kernels, which make up only a small portion of the software code. It is sometimes advantageous to implement the critical kernels using SPPs. This is mostly due to the speedup of the software application and other possible benefits such as reduced energy consumption.

In the study, an embedded system that executes one single threaded software application with N critical kernels is considered. The kernels can either be implemented as software on a GPP or as a SPP using an ASIC or FPGA. The FPGA itself can either be statically or dynamically configured. The rest of the software application, which consists of everything except the critical kernels, is always executed on the GPP. The diagrams of the different system architectures resulting from how the critical kernels are implemented are shown in Figure 3-1. Each system consists of a volatile main memory and a non-volatile memory. In the study the main memory is a random access memory (RAM) and the non-volatile memory is a flash memory. The FPGA is considered to be static RAM (SRAM) based. After the system is powered on, the FPGA requires configurations to be loaded from an external non-volatile memory. In the study, this memory is considered to be a Read Only Memory (ROM).

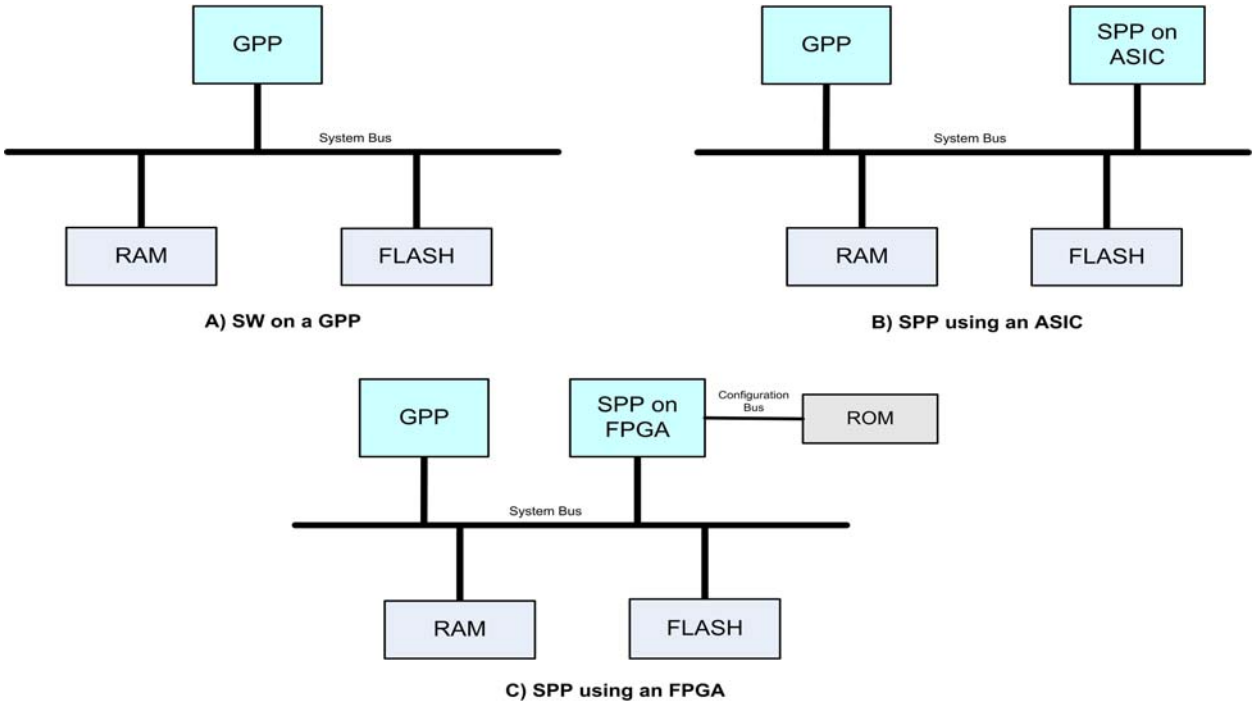

Figure 3-1: Kernel Implementation System Architectures

First, the software application's execution time for the following three design alternatives is examined to analyze the effects of hardware/software partitioning.

1. $\boldsymbol{SW_{system}}$ – System with all critical kernels implemented as software on a GPP. This design alternative has the system architecture show in Figure 3-1-A.
2. $\boldsymbol{DHW_{system}}$ – System with all critical kernels implemented as a SPP on a non-configurable, dedicated hardware (DHW). In the study, the DHW corresponds to the ASIC implementation. This design alternative has the system architecture shown in Figure 3-1-B.
3. $\boldsymbol{CHW_{stc\_system}}$– System with all critical kernels implemented as a SPP on configurable hardware (CHW). The system is statically configured, which means that the FPGA is large enough to have all SPPs configured at all times. In the study, the CHW corresponds to the FPGA implementation. This design alternative has the system architecture shown in Figure 3-1-C.

Before the application's execution time for these three design alternatives can be determined, some of the timing parameters and terms used in the study are defined in Table 3-1.

Table 3-1: Hardware/Software Partitioning Timing Parameters

Term	Description
N	Total number of critical kernels in the software application
n	Denotes a critical kernel out of the total N kernels
$\boldsymbol{SW_n}$	software implementation of critical kernel n
$\boldsymbol{DHW_n}$	ASIC implementation of critical kernel n
$\boldsymbol{CHW_n}$	FPGA implementation of critical kernel n
t_{total}	Total execution time for entire software application using the various design alternatives
t_{code}	Execution time of the entire software application, when all critical kernels are implemented in software, without any hardware acceleration
t_{SW_n}	Average execution time of SW_n executing on the GPP
t_{DHW_n}	Average execution time of DHW_n
t_{CHW_n}	Average execution time of CHW_n
$t_{start_{DHW_n / CHW_n}}$	Average time it takes the GPP to initialize the SPP, before the SPP starts executing. This includes all communication overhead such as parameter passing.
$t_{finish_{DHW_n / CHW_n}}$	Average time it takes the GPP to communicate with the SPP, after the SPP completes execution. This includes all communication overheads such as passing back results to the GPP.
NOE_n	Number of times SW_n executes during execution of the entire software application
R_{SW_n}	Ratio of $t_{SW_n} \times NOE_n$ (the entire runtime of critical kernel n in software) to t_{code} $$R_{SW_n} = \frac{t_{SW_n} \times NOE_n}{t_{code}} \quad \textbf{(1)}$$
R_{total}	Ratio of the entire runtime of all critical kernels in software to t_{code} $$R_{total} = \sum_{n=1}^{N} R_{SW_n} \quad \textbf{(2)}$$
S_{SW_n / DHW_n}	Speedup of executing critical kernel n using DHW_n versus using SW_n $$S_{SW_n / DHW_n} = \frac{t_{SW_n}}{t_{DHW_n}} \quad \textbf{(3)}$$
S_{SW_n / CHW_n}	Speedup of executing critical kernel n using CHW_n versus using SW_n $$S_{SW_n / CHW_n} = \frac{t_{SW_n}}{t_{CHW_n}} \quad \textbf{(4)}$$

Figure 3-2 illustrates the timing that arises for the different design alternatives when one of the critical kernels needs to be executed by the software application.

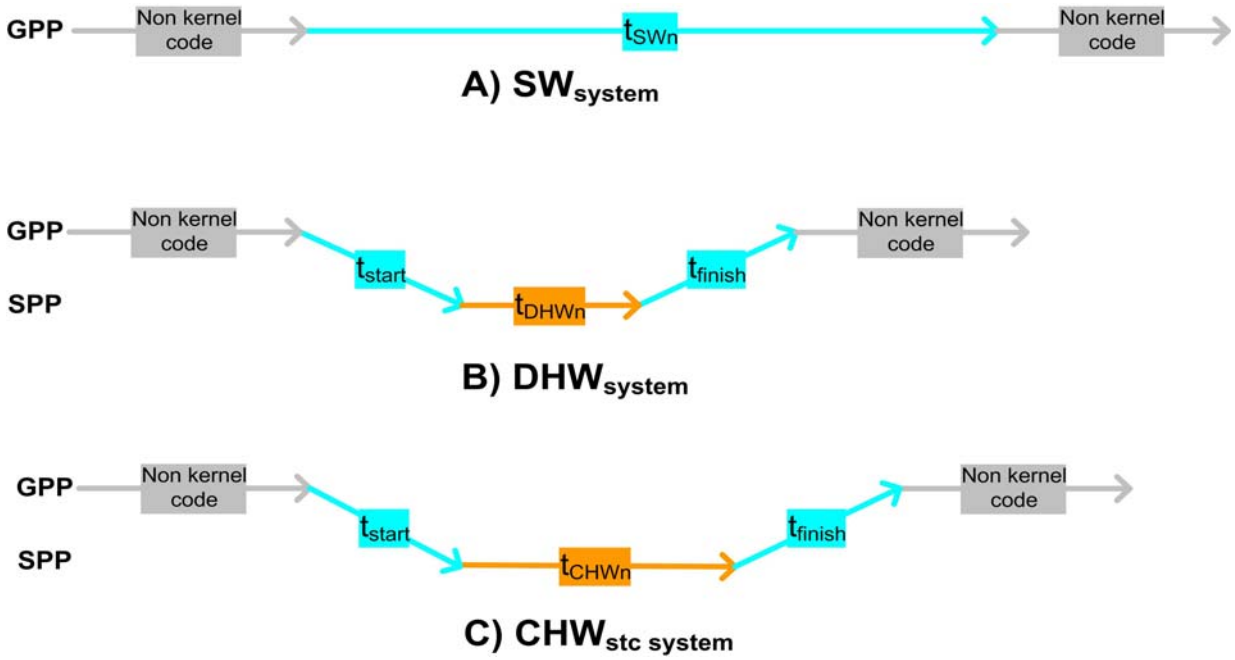

Figure 3-2: Hardware/Software Partitioning Timing Diagram

The SW_{system} design alternative simply executes the kernel using the software implementation. The DHW_{system} and CHW_{stc_system} design alternatives execute the kernel using a SPP implemented on an ASIC or FPGA respectively. By executing the kernel using a SPP, often a shorter execution time is obtained. Since the ASIC implementation of a SPP is faster than an FPGA implementation, the DHW_{system} design alternative has a shorter execution time than the CHW_{stc_system} design alternative. In general, the following relationship can be observed for all critical kernels. It should be noted that when considering execution time the cost of the overheads should also be considered.

$$t_{SW_n} > t_{CHW_n} > t_{DHW_n} \qquad \textbf{(5)}$$

Critical kernels can execute multiple times during the entire execution time of the software application. Using the timing behavior observed in Figure 3-2 and the parameters defined in Table 3-1 Equation 6 was derived to calculate t_{total} for the design alternatives.

$$t_{total} = t_{code}\left(1 - R_{total}\right) + t_{comm\ overhead} + t_{HW\ execution} \qquad \textbf{(6)}$$

$t_{comm\ overhead}$ is the total time spent for the communication between GPP and SPPs. $t_{HW\ execution}$ is the total time the SPPs spend executing. Table 3-2 shows how these two terms can be calculated for each of design alternatives.

Table 3-2: Equation 6 Timing Parameters

System	Equations
***DHW*$_{system}$**	$t_{comm\ overhead} = \sum_{n=1}^{N}\left[\left(t_{start\ DHW_n} + t_{finish\ DHW_n}\right)\times NOE_n\right]$ $t_{HW\ execution} = \sum_{n=1}^{N}\left[t_{DHW_n} \times NOE_n\right]$ $= \sum_{n=1}^{N}\left[R_{SW_n} \times t_{code} \times \frac{1}{S_{SW_n/DHW_n}}\right]$ **(7)**
***CHW*$_{stc\_system}$**	$t_{comm\,overhead} = \sum_{n=1}^{N}\left[\left(t_{start\ CHW_n} + t_{finish\ CHW_n}\right)\times NOE_n\right]$ $t_{HW\,execution} = \sum_{n=1}^{N}\left[t_{CHW_n} \times NOE_n\right]$ $= \sum_{n=1}^{N}\left[R_{SW_n} \times t_{code} \times \frac{1}{S_{SW_n/CHW_n}}\right]$ **(8)**

3.2 On Demand Configuration Scheduling Algorithm

When introducing dynamic reconfiguration in Chapter 2, the configuration scheduling algorithm was defined as the algorithm that determines which configuration is used for FPGA reconfiguration. It was pointed out that a simple on demand configuration scheduling algorithm is currently applied in most systems that use statically generated configurations. This section examines how a dynamically reconfigured system that utilizes an on demand configuration scheduling algorithm, effects the execution

time of the previously described software application. This design alternative is denoted as CHW_{od_system}. This design alternative has a similar architecture to the CHW_{stc_system} design alternative that was illustrated previously in Figure 3-1-C. The main difference of the CHW_{od_system} design alternative is that a smaller FPGA is used to increase the functional density of the system. This smaller FPGA can only contain the configuration of one critical kernel's SPP at a time and dynamic reconfiguration is used during the execution of the application to reconfigure the FPGA with the required precompiled configurations. The on demand configuration scheduling algorithm simply reconfigures the FPGA at runtime, whenever a new configuration is needed and is found not to be configured. The codes of all the critical kernels in the software application are replaced with a call to a configuration scheduling algorithm function. This includes a parameter that indicates the kernel the software application is requesting to execute. The pseudo code of the on demand configuration scheduling algorithm is presented in Algorithm 3-1 and the functions in the pseudo code are explained later.

```
/*kernel_required is the kernel the software application is requesting to
execute*/
OD_CSA(kernel_required)
{
    if( kernel_required is not equal to currently_configured_kernel () )
    {
        initiate_reconfiguration(kernel_required)
        while(kernel_required is not equal to currently_configured_kernel () )
        {do nothing}
    }
    starting_communication_with_SPP(kernel_required))
    wait_until_SPP_is_done_execution()
    finishing_communication_with_SPP (kernel_required))
}
```

Algorithm 3-1: On Demand Configuration Scheduling Algorithm Pseudo Code

currently_configured_kernel()

This function returns the kernel whose SPP is currently configured in the FPGA. It is assumed that the GPP reads from a register in the FPGA to determine which kernel's configuration is currently configured in the FPGA. The execution time of the entire if-statement, which includes a call to this function is defined as t_{check}. This if-statement checks if the required kernel is currently configured.

initiate_reconfiguration(kernel)

This function initiates the reconfiguration of the FPGA with a new kernel's configuration. In this study it is assumed that the FPGA can be reconfigured without any help from the GPP. Most commercial FPGAs such as the Xilinx Virtex II have this ability by allowing the FPGA to act as the master on the configuration bus [Xil06a]. It is assumed that the GPP writes to a register in the FPGA to set the kernel configuration to use for reconfiguration and to initiate the reconfiguration process of the FPGA. The GPP only initiates the reconfiguration process of the FPGA and is free to do any type of computation, while the FPGA is being reconfigured. The execution time of this function is called $t_{initiate}$, which is the time required to initiate the reconfiguration of the FPGA.

starting_communication_with_SPP(kernel)

This function initializes the SPP, before the SPP starts executing. This includes all communication such as passing parameters to the SPP. The execution time of this function is $t_{start_{CHW_n}}$.

wait_until_SPP_is_done_execution()

This function simply waits until the SPP finishes executing. In this study, it is assumed that the GPP goes into a power down mode while the SPP executes and the completion of the SPP is communicated to the GPP via an interrupt. The execution time of this function is t_{CHW_n}.

finishing_communication_with_SPP (kernel)

This function communicates with the SPP, after the SPP completes execution. This includes all communication such as reading back results from the SPP. The execution time of this function is $t_{finish_{CHW_n}}$.

A while-loop is used to wait until the FPGA is reconfigured with the required kernel's SPP. The execution time of the entire while-loop is called $t_{config_{CHW_n}}$, which is the time required to reconfigure the FPGA with CHW_n.

Figure 3-3 illustrates how the on demand configuration scheduling algorithm presented in Algorithm 3-1 behaves. It shows the two different scenarios that can arise for the CHW_{od_system} design alternative, when one of the critical kernels needs to be executed by the software application.

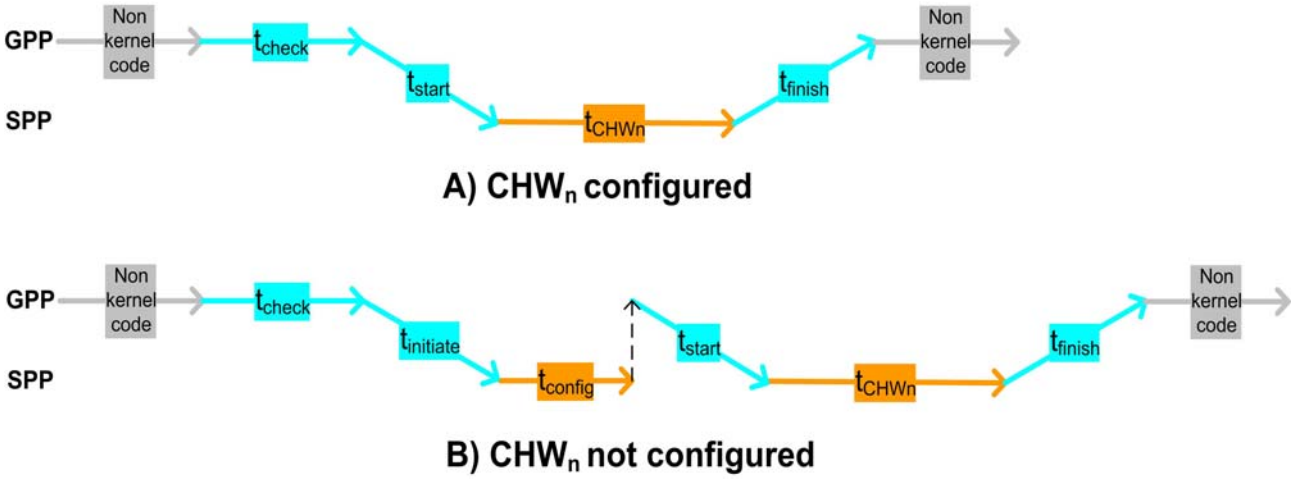

Figure 3-3: On Demand Configuration Scheduling Algorithm

Figure 3-3-A shows the timing for the scenario when the required configuration is already configured in the FPGA. The timing for this scenario is similar to the timing of the CHW_{stc_system} design alternative. Figure 3-3-B shows the timing for the scenario when the required configuration is not configured in the FPGA. In this scenario the on demand configuration scheduling algorithm first initiates the reconfiguration of the FPGA and then waits until the reconfiguration is completed. Afterwards the critical kernel is executed using the SPP configured on the FPGA. Whenever a critical kernel n is required, it causes the scenario in Figure 3-3-B to occur with a probability, which is defined as $P_{CHW_n\ not\ cfg_{ODCSA}}$. The probability of the scenario in Figure 3-3-A then simply becomes $1 - P_{CHW_n\ not\ cfg_{ODCSA}}$. Using the timing behavior observed in Figure 3-3 Equation 9 was derived to calculate t_{total} for the CHW_{od_system} design alternative.

$$t_{total} = t_{code}\left(1 - R_{total}\right) + t_{check}\sum_{n=1}^{N}\left[NOE_n\right] + \sum_{n=1}^{N}\left[P_{CHW_n\ not\ cfg_{ODCSA}} \times \left(t_{config_{CHW_n}} + t_{initiate}\right) \times NOE_n\right] + t_{comm\ overhead} + t_{HW\ execution} \quad \textbf{(9)}$$

where

$$t_{comm\ overhead} = \sum_{n=1}^{N}\left[\left(t_{start_{CHW_n}} + t_{finish_{CHW_n}}\right) \times NOE_n\right]$$

$$t_{HW\ execution} = \sum_{n=1}^{N}\left[R_{SW_n} \times t_{code} \times \frac{1}{S_{SW_n / CHW_n}}\right] = \sum_{n=1}^{N}\left[t_{CHW_n} \times NOE_n\right]$$

It was pointed out previously that the benefit of using CHW_{od_system} over CHW_{stc_system} is the fact that the former has a higher functional density due to the smaller FPGA. The smaller FPGA is advantageous since it results in reduced unit cost, power and area. It is now possible to compare the CHW_{od_system} and CHW_{stc_system} design alternatives by looking at the software application's execution time of both systems. The CHW_{od_system} design alternative has a longer execution time since it has some extra terms in its equation for t_{total} . These terms are shown in Equation 10, which simply subtracts Equation 6 from Equation 9.

$$t_{total_{CHW_{od\_system}}} - t_{total_{CHW_{stc\_system}}} = t_{check}\sum_{n=1}^{N}\left[NOE_n\right] + \sum_{n=1}^{N}\left[P_{CHW_n\ not\ cfg_{ODCSA}} \times \left(t_{config_{CHW_n}} + t_{initiate}\right) \times NOE_n\right] \quad \textbf{(10)}$$

It is reasonable to neglect $t_{initiate}$ and t_{check} in Equation 10, since it can be assumed that these terms are much smaller than $t_{config_{CHW_n}}$. By doing so Equation 11 is obtained, which shows the main factors that contributes to the difference in execution time between the CHW_{od_system} and CHW_{stc_system} design alternatives.

$$t_{total\ CHW\ od\_system} - t_{total\ CHW\ stc\_system} \approx \sum_{n=1}^{N} \left[P_{CHW_n\ not\ cfg\ ODCSA} \times t_{config\ CHW_n} \times NOE_n \right] \quad \textbf{(11)}$$

Equation 11 shows that $t_{config_{CHW_n}}$ is one of the main problems of the on demand configuration scheduling algorithm, which confirms the results in the current literature as pointed out in Chapter 2 as the reconfiguration time overhead. If $t_{config_{CHW_n}}$ cannot be kept short, it increases the execution time of the CHW_{od_system} design alternative over the execution time of the CHW_{stc_system} design alternative to a point where the performance decrease of the former outweighs its benefits of providing higher functional density. This reconfiguration time overhead is especially a problem in embedded systems where tight timing constraints must be met. The reconfiguration time is a function of the architecture and size of the FPGA but the FPGA architectures used currently don't provide a short reconfiguration time [Tes05]. It is also not possible to reduce the size of the FPGA beyond a point where the hardware design of one of the SPPs becomes too large for the FPGA.

From Equation 11 it can be seen that another approach to reduce the main difference in execution time between the CHW_{od_system} and CHW_{stc_system} design alternatives is to reduce $P_{CHW_n\ not\ cfg\ ODCSA}$. Reducing $P_{CHW_n\ not\ cfg\ ODCSA}$ corresponds to reducing the frequency of reconfiguration (FRC), which was already pointed out as a possible technique to reduce the negative effects of the reconfiguration time overhead in Chapter 2. It was also pointed out that due to the inherent inflexibility of the on demand configuration scheduling algorithm it is not possible to reduce the frequency of reconfiguration. The on demand configuration scheduling algorithm has no other alternative than to reconfigure the FPGA with the required configuration, when a configuration is needed and is found not to be currently configured in the FPGA. In the case of the CHW_{od_system} design alternative, the frequency of reconfiguration is only a function of the behavior of the software application. A software application with a high frequency of

different critical kernels being executed after another will have a high frequency of reconfiguration. This is due to the fact that the on demand configuration scheduling algorithm has no alternative than to reconfigure the FPGA whenever the previously executed critical kernel differs from the current one. In order to address these weaknesses of the on demand configuration scheduling algorithm a new type of configuration scheduling algorithm is proposed in the next section.

3.3 Configuration Scheduling Algorithms for DHSP

In the previous section, two of the major problems of the on demand configuration scheduling algorithm were pointed out. The first problem is the reconfiguration time overhead and the second one is the inherent inflexibility of the on demand configuration scheduling algorithm, which makes it impossible to reduce the frequency of reconfiguration. In this section, a configuration scheduling algorithm for performing DHSP is proposed to address the weaknesses of the on demand configuration scheduling algorithm. First the benefits of the DHSP configuration scheduling algorithm are illustrated and then it is examined how the DHSP configuration scheduling algorithm affects the execution time of the previously described software application.

In Chapter 2, the approach in [SL03] and [LV04], which performs DHSP using dynamically generated configurations, was studied. When compared to the on demand configuration scheduling algorithm this approach has the advantage of being able to almost eliminate the reconfiguration time overhead. The ability of this approach to execute the critical kernel using both software and a SPP, allows for the reconfiguration of the FPGA to be performed concurrently, while the GPP executes the kernels in software without any hardware acceleration. Due to this fact the reconfiguration time becomes a less crucial factor to the performance of the system. The flexibility of being able to execute a critical kernel using software or using a SPP also makes it possible to adjust the frequency of reconfiguration. The major drawback of this approach is that it requires profiling and synthesis tool access at runtime.

When developing the DHSP configuration scheduling algorithm it was noted that the on demand configuration scheduling algorithm uses statically generated configurations and that the approach in [SL03] and [LV04] performs DHSP using dynamically generated configurations. It was then realized that DHSP can be further categorized by differentiating between statically and dynamically generated configurations as shown below:

- Hardware/Software Partitioning

    1. Static Partitioning performed at design time
    2. Dynamic Partitioning performed at runtime
        a. Static generation – Precompiled configurations
        b. Dynamic generation – Configurations generated at runtime

It is then proposed that it is possible to develop a configuration scheduling algorithm to perform DHSP using statically generated configurations. This approach combines the advantages and eliminates the major drawbacks of both the on demand configuration scheduling algorithm and the approaches studied in [SL03] and [LV04]. The DHSP configuration scheduling algorithm eliminates the need for profiling and synthesis tool access at runtime, by using statically generated configurations. By performing DHSP, the configuration scheduling algorithm can also eliminate the reconfiguration time overhead and reduce the frequency of reconfiguration. Next these points are explained in further detail.

In the study DHSP is the procedure of determining at runtime whether a critical kernel should be executed using software or using a SPP. A design alternative is proposed, which utilizes a new configuration scheduling algorithm to perform DHSP using statically generated configurations. This design alternative has the same system architecture as the CHW_{od_system} design alternative, which was explained previously. Therefore, it provides the same benefit of increased functional density from a smaller FPGA. And as for the CHW_{od_system} design alternative this smaller FPGA can only contain the configuration of one critical kernel's SPP at a time. But the DHSP configuration scheduling algorithm is not as simple as the on demand configuration scheduling algorithm and does not simply reconfigure the FPGA at runtime, whenever a new configuration is needed. Instead, whenever the software application needs to execute a critical kernel, the configuration scheduling algorithm is responsible to select at runtime if it should be executed in software or using the SPP. The ability of this approach to execute the critical kernel using both the software and SPP, allows for the reconfiguration of the FPGA to be performed concurrently while the GPP executes the software without any hardware acceleration. This is illustrated in Figure 3-4 by comparing the scenario where a required configuration is not found in the FPGA.

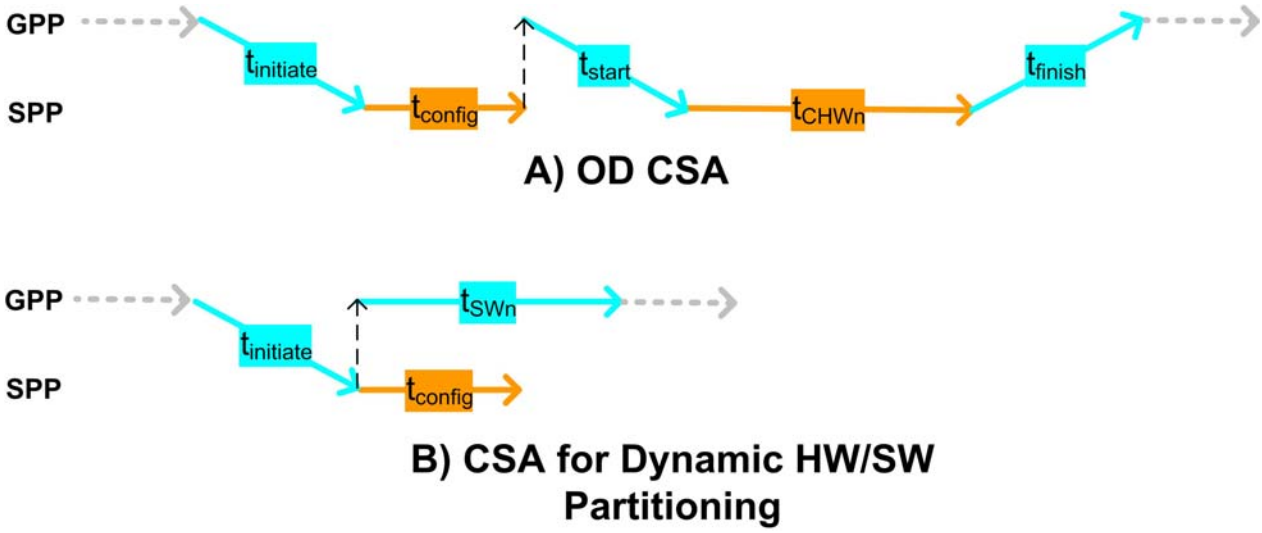

Figure 3-4: Concurrent Reconfiguration

The on demand configuration scheduling algorithm is required to wait until the FPGA is reconfigured before it can proceed with executing the kernel using the SPP. The DHSP configuration scheduling algorithm, on the other hand, has the ability to initiate the reconfiguration of the FPGA and then proceed with the execution of the kernel in software. The reconfiguration time therefore becomes a less crucial factor for the DHSP configuration scheduling algorithm than for the on demand configuration scheduling algorithm, due to the ability of being able to execute the critical kernel using both the software and SPP. Reconfiguration of the FPGA with a configuration of a SPP occurs here due to the anticipation of the SPP being required by the software application in the near future. In general, the major phases of a configuration scheduling algorithm can be categorized into the following:

- Configuration Scheduling Algorithm Phases

    1. Monitoring – Gathering information about the software application's behavior at runtime
    2. Selection – Deciding which configuration should be configured in the FPGA, using the gathered information from the monitoring phase
    3. Reconfiguration – Performing the actual reconfiguration of the FPGA with the required configurations

The on demand configuration scheduling algorithm does not have a monitoring and selection phase, since it simply configures the FPGA with the currently required configuration. DHSP configuration

scheduling algorithms on the other hand have the ability to execute critical kernels either in software or hardware and can therefore be developed such that the monitoring and selection phases are tuned and optimized to the behavior of the software application at hand. In contrast, the same on demand configuration scheduling algorithm is used for all software applications regardless of their behavior. This is due to the fact that the inflexibility of the on demand configuration scheduling algorithm does not allow customization and tuning to target a specific software application. DHSP configuration scheduling algorithms, on the other hand, are flexible and can be developed such that they are tuned to the behavior of the software application. Here the behavior of the software application that is of interest is the relation between critical kernels, which affects when and how critical kernels execute. Section 3.1 examines the $SW_{system,}$ DHW_{system}, CHW_{stc_system} and CHW_{od_system} design alternatives and analyzes how the execution time of the software application is affected. The behavior of the software application is irrelevant to the analysis of the execution time for the on demand configuration scheduling algorithm. For design alternatives that use DHSP configuration scheduling algorithms, it is necessary to specify the behavior of the software application since the implementation of this configuration scheduling algorithm depends on it.

3.3.1 Temporal Locality Configuration Scheduling Algorithm

This section looks at a behavior of the software application introduced in Section 3.1, in which the critical kernels are more likely to execute in the near future if they executed frequently in the near past. For the purpose of this thesis the DHSP configuration scheduling algorithm targeting this behavior of the application is called the temporal locality (TL) configuration scheduling algorithm and this design alternative is denoted as CHW_{tl_system}. This design alternative has the same system architecture as the CHW_{od_system} design alternative. First, the behavior of the application is introduced using the following example. When a cell phone that is used to play music receives a phone call, it goes through the sequence of operational modes of just playing music, having a phone conversation and back to just playing music. Kernel 1 is used for decoding music files when playing music and kernel 2 is used for decrypting the incoming voice messages of the phone conversation. Kernel 1 only executes when playing music and kernel 2 only executes when the phone conversation is taking place. Here, both critical kernels are more likely to execute in the near future if they executed frequently in the near past. For example, kernel 2 starts executing a couple of times shortly after the phone conversation starts and kernel 1 stops executing since the music is stopped. The fact that the kernel 1 executes less frequently at this point indicates that it is not likely to execute in the near future. Also, the fact that kernel 2 executes frequently at this point

indicates that it likely to execute in the near future. This software application has the characteristic that different kernels are more dominant and execute more frequently in different operational modes.

A DHSP configuration scheduling algorithm can take advantage of this behavior of the application to accurately predict and schedule configurations of SPP implementations of these critical kernels. In this study, the DHSP configuration scheduling algorithm targeting this type of application is called the temporal locality (TL) configuration scheduling algorithm, after a similar phenomenon that is taken advantage of in memory caching. Also software applications with this type of behavior are defined as applications with kernels that have a temporal locality characteristic. In order to illustrate the possible advantages of the temporal locality configuration scheduling algorithm over the on demand configuration scheduling algorithm the following sequence of critical kernel executions from a software application with a temporal locality characteristic is used:

kernel 1→ kernel 2 → kernel 1→ kernel 1 → kernel 3 → kernel 1 → kernel 2 → kernel 1

It can be seen that kernel 1executes most frequently during the sequence. If it is assumed that the FPGA is initially configured with kernel 1, the on demand configuration scheduling algorithm would reconfigure the FPGA six times for this sequence of critical kernels. Each kernel would be executed using their corresponding SPP. A temporal locality configuration scheduling algorithm implementation, on the other hand, could execute the entire sequence without ever reconfiguring the FPGA. If the temporal locality configuration scheduling algorithm had predicted that kernel 1 will be used frequently in the upcoming sequence it could have left kernel 1 configured in the FPGA for the entire sequence. Here, only kernel 1 would be executed using the SPP, while the other kernels would be executed in software. This example illustrates that the flexibility of the temporal locality configuration scheduling algorithm can be used to reduce the frequency of reconfiguration significantly when compared to the on demand configuration scheduling algorithm. The temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm, by only trying to execute the most frequently executing kernel using its SPP while executing the other kernels using their software implementation. The on demand configuration scheduling algorithm on the other hand aims at executing all the kernels using their SPP, which results in a higher frequency of reconfiguration. Next, it is explained how the temporal locality configuration scheduling algorithm was designed and afterwards an analysis is presented on how the execution time of the software application is affected by the use of this configuration scheduling algorithm.

A temporal locality configuration scheduling algorithm could take advantage of the temporal locality characteristic of an application by simply keeping track of how many times each kernel has executed in the recent past during the monitoring phase. Whenever a critical kernel's SPP is needed and is currently configured in the FPGA, the configuration scheduling algorithm simply uses the SPP to execute the kernel. Whenever a critical kernel's SPP is needed and is found not to be configured in the FPGA, the configuration scheduling algorithm goes into the selection phase. The selection phase determines whether the FPGA should be reconfigured with the currently required SPP. By looking at the information of recently executed kernels, it is possible to determine if the currently required SPP is likely to be needed in the near future. Whether reconfiguration is determined to be required or not, in this case the critical kernel is always executed in software. In the situation when reconfiguration of the FPGA with the currently required SPP is determined to be required, the temporal locality configuration scheduling algorithm initiates the reconfiguration and executes the critical kernel in software. By doing so there is no need to wait until the FPGA is reconfigured to execute the critical kernel. Once the FPGA is reconfigured and the same critical kernel executes at some time in the near future, it is possible to execute it using the SPP. In the situation when reconfiguration of the FPGA is determined not to be required, the temporal locality configuration scheduling algorithm simply executes the critical kernel in software without initiating any reconfiguration.

The code of all the critical kernels in the software application is replaced with a call to the temporal locality configuration scheduling algorithm function, which includes a parameter that indicates the kernel the software application is requesting to execute. The temporal locality configuration scheduling algorithm then decides whether to execute the critical kernel in software or using the SPP. The pseudo code of the temporal locality configuration scheduling algorithm is presented in Algorithm 3-2 and the functions in the pseudo code, which were not explained with the introduction of Algorithm 3-1, are explained thereafter.

```
/*kernel_required is the kernel the software application is requesting to
execute*/
TL_CSA(kernel_required)
{
    update_history_tl_csa(kernel_required) //Monitoring Phase
    if( kernel_required is not equal to currently_configured_kernel () )
    {
        if( reconfiguration_required_tl_csa() ) //Selection Phase
        {
            initiate_reconfiguration(kernel_required)
        }
        execute_in_SW(kernel_required)
    }
    else     // kernel_required is already configured
    {
        starting_communication_with_SPP(kernel_required)
        wait_until_SPP_is_done_execution()
        finishing_communication_with_SPP (kernel_required)
    }
}
```

Algorithm 3-2: Temporal Locality Configuration Scheduling Algorithm Pseudo Code

update_history_tl_csa(kernel_required)

This function simply gathers information about the software application that might be useful in the selection phase. During this monitoring phase the temporal locality configuration scheduling algorithm simply keeps track of how many times each kernel has executed in the recent past. The average execution time of this function is called $t_{update_{TLCSA}}$.

reconfiguration_required_tl_csa()

During this selection phase, the function decides which configuration should be configured in the FPGA using the gathered information from the monitoring phase. By looking at the information of recently executed kernels, it determines if the currently required kernel's SPP is likely to be needed in the near future and whether it should reconfigure the FPGA with the currently required kernel's

configuration. The function returns a Boolean value indicating whether reconfiguration is required or not. The average execution time of the entire second if-statement, which includes a call to this function, is called $t_{selection\_TLCSA}$ (since the number of kernels are constant in an application, this time does not depend on the number of kernels). More details about a possible implementation of the selection phase are presented in Chapter 4.

execute_in_SW(kernel_required)

This function simply executes the kernel using its software implementation. The execution time of this function is t_{SW_n}.

Figure 3-5 illustrates how the temporal locality configuration scheduling algorithm presented in Algorithm 3-2 behaves and it is used to analyze how the execution time of the software application introduced in Section 3.1 is affected by the use of the temporal locality configuration scheduling algorithm. It shows the three different scenarios that can arise for the CHW_{tl_system} design alternative, when one of the critical kernels needs to be executed by the software application.

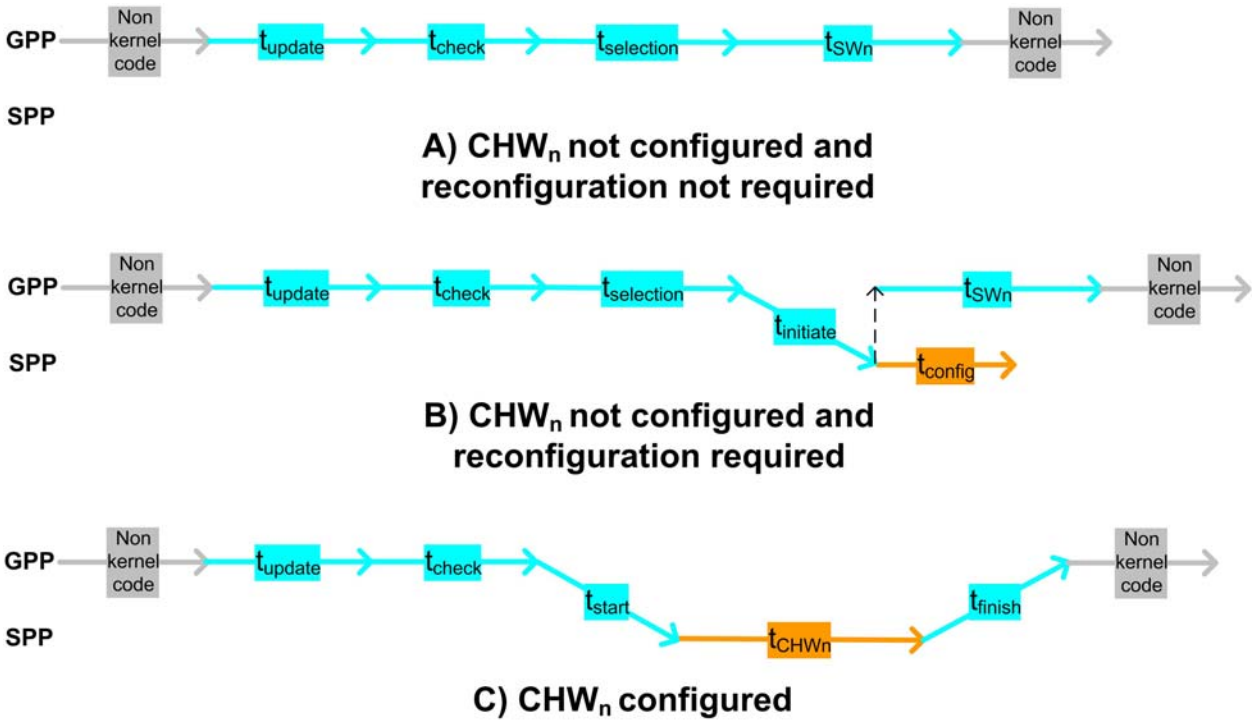

Figure 3-5: Temporal Locality Configuration Scheduling Algorithm

Figure 3-5.A and Figure 3-5.B show the timing for the scenario when the required configuration is not configured in the FPGA. In the scenario in Figure 3-5.A, the configuration scheduling algorithm determines that reconfiguration is not required and the kernel is simply executed in software without initiating any reconfiguration of the FPGA. In the scenario in Figure 3-5.B, the configuration scheduling algorithm determines that a reconfiguration is required. Here, the temporal locality configuration scheduling algorithm first initiates the reconfiguration of the FPGA with the currently required configuration and then executes the current kernel in software. By doing so, there is no need to wait until the FPGA is reconfigured in order to execute the critical kernel. Once the FPGA is reconfigured and the same critical kernel executes at some time in the near future, it is possible to execute it using the SPP. This scenario is shown in Figure 3-5.C where the required configuration is already configured in the FPGA. Here the critical kernel is simply executed using the SPP configured on the FPGA. In this scenario, no selection phase is required since the FPGA is already configured with the currently required configuration and therefore no reconfiguration is needed.

Whenever a critical kernel n is required, it causes the scenario in Figure 3-5.A or Figure 3-5.B to occur with a probability, which is defined as $P_{CHW_n\ not\ cfg_{TLCSA}}$. This is the probability of the configuration of critical kernel n not being configured when it is required. The probability of the scenario in Figure 3-5.C then simply becomes $1 - P_{CHW_n\ not\ cfg_{TLCSA}}$. Whenever a critical kernel n is required, it causes the scenario in Figure 3-5.B to occur with a probability, which is defined as $P_{CHW_n\ reconfig_{TLCSA}}$.This is the probability that the FPGA will be reconfigured. Using the timing behavior observed in Figure 3-5, Equation 12 was derived to calculate t_{total} for the CHW_{tl_system} design alternative.

$$
\begin{aligned}
t_{total} \;=\; & t_{code}\left(1 - R_{total}\right) + \left(t_{check} + t_{update_{TLCSA}}\right)\sum_{n=1}^{N}\left[NOE_n\right] + \\
& \sum_{n=1}^{N}\left[P_{CHW_n\ not\ cfg_{TLCSA}} \times \left(t_{selection_{TLCSA}} + t_{SW_n}\right) \times NOE_n\right] + \\
& \sum_{n=1}^{N}\left[P_{CHW_n\ reconfig_{TLCSA}} \times t_{initiate} \times NOE_n\right] + \\
& t_{comm\ overhead} + t_{HW\ execution}
\end{aligned}
\qquad \textbf{(12)}
$$

where

$$t_{comm\ \ overhead} = \sum_{n=1}^{N} \left[\left(1 - P_{CHW_n\ not\ cfg_{TLCSA}}\right) \times \left(t_{start_{CHW_n}} + t_{finish_{CHW_n}}\right) \times NOE_n \right]$$

$$t_{HW\ \ execution} = \sum_{n=1}^{N} \left[\left(1 - P_{CHW_n\ not\ cfg_{TLCSA}}\right) \times t_{CHW_n} \times NOE_n \right]$$

The CHW_{tl_system} and CHW_{od_system} design alternatives can now be compared by looking at the software application's execution time of both systems. From Equation 12 it can be seen that the execution time of the CHW_{tl_system} design alternative does not directly depend on $t_{config_{CHW_n}}$. The execution time of the CHW_{od_system} design alternative on the other hand does depend on it as shown in Equation 9. This shows that the reconfiguration time overhead is a less crucial factor for the CHW_{tl_system} design alternative than for the CHW_{od_system} design alternative due to the alternate execution path of critical kernels using software that exists when using the temporal locality configuration scheduling algorithm. This enables the CHW_{tl_system} design alternative to be useful in embedded systems where tight worst case timing constraints must be met. When using the CHW_{tl_system} design alternative, it also becomes less critical if the FPGA has a long reconfiguration time due to its size or architecture.

From Equation 12 it can also be seen that it is desirable to reduce $P_{CHW_n\ not\ cfg_{TLCSA}}$ to increase the frequency of executing critical kernels using their SPP implementation. $P_{CHW_n\ not\ cfg_{TLCSA}}$ is not only a function of the behavior of the software application, which is not true for the CHW_{od_system}. Here $P_{CHW_n\ not\ cfg_{TLCSA}}$ also depends on the accuracy with which the temporal locality configuration scheduling algorithm implementation predicts the configuration that will be required most frequently in the near future. $P_{CHW_n\ reconfig_{TLCSA}}$ corresponds to the frequency of reconfiguration (FRC) of the CHW_{tl_system} design alternative. As shown previously, the temporal locality configuration scheduling algorithm's ability to execute the critical kernels using software or the SPP can be used to reduce and adjust the frequency of reconfiguration, which is not possible with the on demand configuration scheduling algorithm.

3.3.2 Kernel Correlation Configuration Scheduling Algorithm

This section looks at a behavior of the software application introduced in Section 3.1, in which the critical kernel's likelihood to execute in the near future depends on which critical kernel executed last. The DHSP

configuration scheduling algorithm targeting this behavior of the application is called the kernel correlation (KC) configuration scheduling algorithm and this design alternative is denoted as CHW_{kc_system}. This design alternative also has the same system architecture as the CHW_{od_system} design alternative.

First, the behavior of the application is introduced using the following example. A handheld device is used to watch a streaming video and afterwards the same device is used to listen to streaming music. Both the music and video data is being received by the device over an encrypted wireless channel. The device therefore goes through the operational modes of just watching video and then just listening to music. Kernel 1 is used for decoding the video file format, kernel 2 is used for decrypting all the incoming data over the wireless channel and kernel 3 is used for decoding the music file format. In the first operational mode of watching video the application decrypts some of the streaming data and then decodes the video format. Therefore kernel 2 is always followed by kernel 1 and vice versa. In the second operational mode of listening to music, the application decrypts some of the streaming data and then decodes the music format. Here kernel 2 is always followed by kernel 3 and vice versa. This application has the characteristic that a critical kernel's likelihood to execute in the near future depends on which critical kernel executed last. A DHSP configuration scheduling algorithm can take advantage of this behavior of the application to accurately predict and schedule configurations of SPP implementations of these critical kernels. In the study the DHSP configuration scheduling algorithm targeting this type of application is named the kernel correlation (KC) configuration scheduling algorithm. Also software applications with this type of behavior are defined as applications with kernels that have a kernel correlation characteristic.

The kernel correlation configuration scheduling algorithm can take advantage of this behavior of the application by simply keeping track of which kernels have executed most frequently after each kernel in the recent past. Whenever a critical kernel's SPP is needed and is found not to be configured in the FPGA, the kernel correlation configuration scheduling algorithm goes into the selection phase similar to the temporal locality configuration scheduling algorithm. The difference of the selection phase of the kernel correlation configuration scheduling algorithm from the temporal locality configuration scheduling algorithm is that it does not solely determine whether the FPGA should be reconfigured with the currently required SPP. Instead, it considers all kernels' SPPs for configuration into the FPGA. By looking at the information of which kernels have executed most frequently after the currently required kernel, it is possible to determine the kernel's SPP, which is most likely to be needed in the near future. This kernel's SPP is then be configured in the FPGA if it is not already configured. Whenever a critical kernel's SPP is needed and is currently configured in the FPGA, the kernel correlation configuration scheduling

algorithm simply uses the SPP to execute the kernel much like the temporal locality configuration scheduling algorithm. The only difference here is that the kernel correlation configuration scheduling algorithm also goes into the selection phase after executing the kernel using the SPP. The selection phase determines the kernel's SPP, which is most likely to be needed in the near future and reconfigures the FPGA with this SPP if it is not already configured. It is noted that the kernel correlation configuration scheduling algorithm is similar to the on demand configuration scheduling algorithm in that it aims at executing all the kernels using their SPP. The temporal locality configuration scheduling algorithm on the other hand tries to execute only the most frequently executing kernel using its SPP while executing the other kernels using their software implementation.

The code of all the critical kernels in the software application is replaced with a call to the kernel correlation configuration scheduling algorithm function, which includes a parameter that indicates the kernel the software application is requesting to execute. The kernel correlation configuration scheduling algorithm then decides whether to execute the critical kernel in software or using the SPP. The pseudo code of the kernel correlation configuration scheduling algorithm is presented in Algorithm 3-3 and the functions in the pseudo code, which were not explained with the introduction of Algorithm 3-1 and Algorithm 3-2, are explained thereafter.

```
/*kernel_required is the kernel the software application is requesting to
execute*/
next_kernel //indicates the next kernel's SPP to be configured into the FPGA

KC_CSA(kernel_required)
{
    update_history_kc_csa(kernel_required) //Monitoring Phase
    if( kernel_required is not equal to currently_configured_kernel () )
    {
        if( reconfiguration_required_kc_csa() ) //Selection Phase
        {// reconfigure FPGA before executing using software
            initiate_reconfiguration(next_kernel)
        }
        execute_in_SW(kernel_required)
    }
    else    // kernel_required is already configured
    {
        starting_communication_with_SPP(kernel_required)
        wait_until_SPP_is_done_execution()
        finishing_communication_with_SPP (kernel_required)

        if( reconfiguration_required_kc_csa() ) //Selection Phase
        {// reconfigure FPGA after executing using SPP
            initiate_reconfiguration(next_kernel)
        }
    }
}
```

Algorithm 3-3: Kernel Correlation Configuration Scheduling Algorithm Pseudo Code

update_history_kc_csa(kernel_required)

This function simply gathers information about the software application that might be useful in the selection phase. During this monitoring phase the kernel correlation configuration scheduling algorithm

simply keeps track of which kernels have executed most frequently after each kernel in the recent past. The average execution time of this function is called $t_{update\ KCCSA}$.

reconfiguration_required_kc_csa()

During this selection phase the function decides which configuration should be configured in the FPGA, using the gathered information from the monitoring phase. The difference of the selection phase of the kernel correlation configuration scheduling algorithm from the temporal locality configuration scheduling algorithm is that it does not only determine whether the FPGA should be reconfigured with the currently required SPP. Instead, it considers all kernels' SPPs for configuration into the FPGA. By looking at the information of which kernels have executed most frequently after the currently required kernel, it is possible to determine the kernel's SPP, which is most likely to be needed in the near future. The function sets this kernel equal to the variable *next_kernel* and returns a Boolean value indicating whether reconfiguration is required or not. Reconfiguration is required if the *next_kernel* is not already configured in the FPGA. The execution time of the entire second if-statement, which includes a call to this function, is called $t_{selection\ KCCSA}$.

Figure 3-6 illustrates how the kernel correlation configuration scheduling algorithm presented in Algorithm 3-3 behaves and it is used to analyze how the execution time of the software application, which was introduced in Section 3.1, is affected by the use of the kernel correlation configuration scheduling algorithm. It shows the four different scenarios that can arise for the CHW_{kc_system} design alternative, when one of the critical kernels needs to be executed by the software application. Figure 3-6.A and Figure 3-6.B show the timing for the scenario when the required configuration is not configured in the FPGA. In the scenario in Figure 3-6.A, the configuration scheduling algorithm determines that reconfiguration is not required and the kernel is simply executed in software without initiating any reconfiguration of the FPGA. In the scenario in Figure 3-6.B, the configuration scheduling algorithm determines that reconfiguration is required. Here the kernel correlation configuration scheduling algorithm first initiates the reconfiguration of the FPGA and then executes the current kernel in software. The scenarios shown in Figure 3-6.C and Figure 3-6.D show the timing for the scenario when the required configuration is already configured in the FPGA. Here the kernel correlation configuration scheduling algorithm simply uses the SPP to execute the kernel much like the temporal locality configuration scheduling algorithm. The only difference here is that the kernel correlation configuration scheduling algorithm also goes into the selection phase after executing the kernel using the SPP. The selection phase determines the kernel's SPP, which is most likely

to be needed next. The FPGA is reconfigured with this SPP if it is not already configured as shown in Figure 3-6.C and Figure 3-6.D.

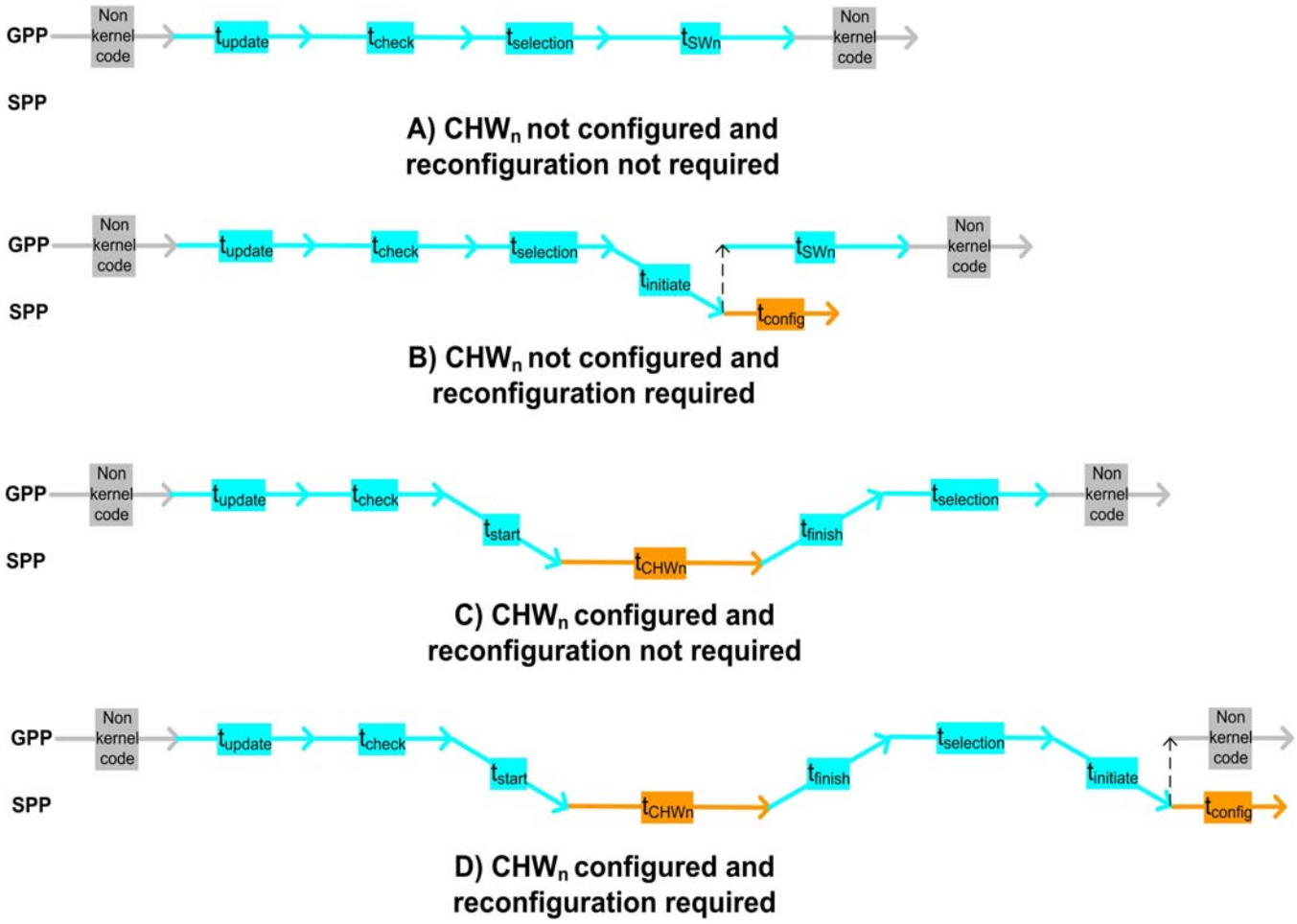

Figure 3-6: Kernel Correlation Configuration Scheduling Algorithm Timing Diagram

Whenever a critical kernel n is required, it causes the scenario in Figure 3-6.A or Figure 3-6.B to occur with a probability, which is defined as $P_{CHW_n\ not\ cfg_{KCCSA}}$. This is the probability of the configuration of critical kernel n not being configured when it is required. The probability of the scenarios in Figure 3-6.C or Figure 3-6.D then simply becomes $1 - P_{CHW_n\ not\ cfg_{KCCSA}}$. Whenever a critical kernel n is required, it causes the scenario in Figure 3-6.B to occur with a probability, which is defined as $P_{CHW_n\ reconfig\ \ notcfg_{KCCSA}}$. Also, whenever a critical kernel n is required, it causes the scenario in Figure

3-6.D to occur with a probability, which is defined as $P_{CHW_n\ reconfig\ \ cfg_{KCCSA}}$. The sum of $P_{CHW_n\ reconfig\ \ cfg_{KCCSA}}$ and $P_{CHW_n\ reconfig\ \ notcfg_{KCCSA}}$ is the probability that the FPGA will be reconfigured whenever critical kernel n is required. Using the timing behavior observed in Figure 3-6, Equation 13 was derived to calculate t_{total} for the CHW_{kc_system} design alternative.

$$
\begin{aligned}
t_{total} \;=\;& t_{code}\left(1 - R_{total}\right) + \\
& \left(t_{check} + t_{update\ \ KCCSA} + t_{selection\ \ KCCSA}\right)\sum_{n=1}^{N}\left[NOE_{\ n}\right] + \\
& \sum_{n=1}^{N}\left[P_{CHW_n\ not\ cfg_{KCCSA}} \times t_{SW_n} \times NOE_{\ n}\right] + \\
& \sum_{n=1}^{N}\left[\left(P_{CHW_n\ reconfig\ \ cfg_{KCCSA}} + P_{CHW_n\ reconfig\ \ notcfg_{KCCSA}}\right) \times t_{initiate} \times NOE_{\ n}\right] \\
& + t_{comm\ \ overhead} + t_{HW\ \ execution}
\end{aligned}
\qquad \textbf{(13)}
$$

where

$$t_{comm\ \ overhead} = \sum_{n=1}^{N}\left[\left(1 - P_{CHW_n\ not\ cfg_{KCCSA}}\right) \times \left(t_{start\ \ CHW_n} + t_{finish\ \ CHW_n}\right) \times NOE_{\ n}\right]$$

$$t_{HW\ \ execution} = \sum_{n=1}^{N}\left[\left(1 - P_{CHW_n\ not\ cfg_{KCCSA}}\right) \times t_{CHW_n} \times NOE_{\ n}\right]$$

It can be seen that it is desirable to reduce $P_{CHW_n\ not\ cfg_{KCCSA}}$ to increase the frequency of executing critical kernels using their SPP implementation. Similar to the CHW_{tl_system} design alternative, $P_{CHW_n\ not\ cfg_{KCCSA}}$ is not only a function of the behavior of the software application but also depends on how accurately the kernel correlation configuration scheduling algorithm implementation predicts the configuration that will be required next. Also similar to the CHW_{tl_system} design alternative, the t_{total} equation does not directly depend on $t_{config_{CHW_n}}$.

It is also noted that the reconfiguration time overhead can still affect the overall execution time of the software application by affecting $P_{CHW_n\ not\ cfg_{KCCSA}}$. A larger reconfiguration time increases $P_{CHW_n\ not\ cfg_{KCCSA}}$, since the FPGA is unusable for a longer period of time due to the longer

reconfiguration process. A large reconfiguration time can cause the SPP of a critical kernel to be not configured in the FPGA on time for it to be used. At this point the kernel is executed using its software implementation instead of its SPP implementation causing an increase in the overall execution time of the software application.

3.4 Summary

This chapter first introduced an embedded system that executes one single threaded software application with N critical kernels. This application was used to examine the effects of hardware/software partitioning by studying the SW_{system}, DHW_{system} and CHW_{stc_system} design alternatives. It was then identified how on demand configuration scheduling algorithms, which are used in many dynamically reconfigurable systems, affect an application's execution time. Then, it was shown that the reconfiguration time overhead is one of the main problems of the on demand configuration scheduling algorithm. It was also pointed out that the frequency of reconfiguration of the CHW_{od_system} design alternative is only a function of the behavior of the software application. A software application with a high frequency of different critical kernels being executed after another will have a high frequency of reconfiguration due to the fact that the on demand configuration scheduling algorithm has no alternative than to reconfigure the FPGA whenever the previously executed critical kernel differs from the current one.

To address these weaknesses of the on demand configuration scheduling algorithm, a new type of configuration scheduling algorithm was then proposed. DHSP configuration scheduling algorithms can be used to perform dynamic hardware/software partitioning using statically generated configurations. This approach combines the advantages and eliminates the major drawbacks of both the on demand configuration scheduling algorithm and the approaches studied in [SL03] and [LV04]. The DHSP configuration scheduling algorithm eliminates the need for profiling and synthesis tool access at runtime, by using statically generated configurations. By performing DHSP, the configuration scheduling algorithm can also reduce the reconfiguration time overhead. It was then pointed out that it is necessary to specify the behavior of the software application for design alternatives that use DHSP configuration scheduling algorithms, since the implementation of these configuration scheduling algorithms depends on it. The temporal locality and kernel correlation behavior of software applications were then analyzed and it was explained how two corresponding DHSP configuration scheduling algorithms were designed that specifically target the two different application types. Afterwards, it was examined how the execution time of the software application is affected by the use of the temporal locality and kernel correlation

configuration scheduling algorithms. Using Equations 12 and 13, it was shown that the execution time of the CHW_{tl_system} and CHW_{kc_system} design alternatives does not directly depend on the reconfiguration time. This is due to the alternate execution path of critical kernels using software that exists for both DHSP configuration scheduling algorithm based design alternatives. This enables both DHSP configuration scheduling algorithm based design alternatives to be useful in systems where timing constraints should not depend on the reconfiguration time. Both DHSP configuration scheduling algorithm based design alternatives guarantee a worst case kernel execution latency that is equivalent to the runtime of the kernel in software and some small additional overhead due to the execution of the configuration scheduling algorithm. The on demand configuration scheduling algorithm on the other hand can only guarantee a worst case kernel execution latency that is dependent on the reconfiguration time. It was noted that the reconfiguration time overhead can still affect the overall execution time of the software application for both DHSP configuration scheduling algorithm based design alternatives since a larger reconfiguration time can cause the FPGA to be unusable for a longer period of time.

It was also shown that the temporal locality configuration scheduling algorithm's ability to execute the critical kernels using software or the SPP can be used to reduce and adjust the frequency of reconfiguration, which is not possible with the on demand configuration scheduling algorithm. When the software application has a high frequency of different critical kernels being executed after another the frequency of reconfiguration is high for the on demand configuration scheduling algorithm. The temporal locality configuration scheduling algorithm is implemented such that it reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. The temporal locality configuration scheduling algorithm accomplishes this by leaving only the most frequently executing kernel's configuration in the FPGA while executing the other kernels using their software implementation. Even when the software application has a high frequency of different critical kernels being executed after another, the frequency of reconfiguration stays low for the temporal locality configuration scheduling algorithm since the FPGA is not reconfigured each time a different critical kernel executes. The kernel correlation configuration scheduling algorithm on the other hand is not implemented to reduce the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. Each time a kernel is required to execute the kernel correlation configuration scheduling algorithm determines which kernel executed the most frequent in the recent past after the currently required kernel. The kernel correlation configuration scheduling algorithm reconfigures the FPGA every time it anticipates that a different kernel is going to execute next after the current kernel. When a software application has a high frequency of different critical kernels being executed after

another and when the kernel correlation configuration scheduling algorithm makes mostly correct predictions, the resulting frequency of reconfiguration becomes as high as the frequency of reconfiguration of the on demand configuration scheduling algorithm. The on demand configuration scheduling algorithm simply reconfigures the FPGA at runtime, whenever a configuration is needed and is found not to be currently configured. The kernel correlation configuration scheduling algorithm reconfigures the FPGA in anticipation of a configuration being needed, which is currently not configured in the FPGA. Therefore, a kernel correlation configuration scheduling algorithm that makes mostly correct predictions has a frequency of reconfiguration, which is as high as the frequency of reconfiguration of the on demand configuration scheduling algorithm. The temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm, by only trying to execute the most frequently executing kernel using its SPP while executing the other kernels using their software implementation. The kernel correlation configuration scheduling algorithm and on demand configuration scheduling algorithm, on the other hand, aim to execute all the kernels using their SPP, which results in a higher frequency of reconfiguration.

The next chapter looks at a case study to analyze the performance of the various configuration scheduling algorithms and design alternatives, introduced in this chapter. The study examines the differences in the execution time of software applications that results from the use of different configuration scheduling algorithms and design alternatives. The effects the reconfiguration time overhead has on the execution time of the software application is also analyzed. Additionally the case study is used to analyze to which extent the temporal locality configuration scheduling algorithm can reduce and adjust the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm.

Chapter 4
Configuration Scheduling Algorithm Performance Evaluation

This chapter first presents the case study used to evaluate the various configuration scheduling algorithms and design alternatives, which were introduced previously. Afterwards the modeling framework and methodology that was chosen to examine the performance of the various design alternatives is explained. After introducing the implementation of the model the simulation results are presented. The results are then examined and it is determined whether they are consistent with the analysis and theory that was presented in Chapter 3. Finally the patterns and trends in the simulation results are identified and the insights gained from the experiments are summarized.

4.1 Case Study

In the rest of the study (Chapter 4 to Chapter 6) the tradeoffs that exist between the following design alternatives are examined:

1. SW_{system}
2. CHW_{stc_system}
3. CHW_{od_system}
4. CHW_{tl_system}
5. CHW_{kc_system}

To determine the tradeoffs, it is required to perform a case study on a problem, which considers implementing a system using one of these design alternatives. In this section, the embedded system that was used as the case study is introduced. For the software application introduced in Section 3.1, MediaBench benchmarks [Med06] were used. The benchmarks consist of a set of multimedia applications designed for embedded systems. These benchmarks were chosen since they were used extensively in research projects, allowing the current study to benefit from the various studies that were already

performed. In [VS02] some of these benchmarks were taken and their critical kernels were identified through profiling. Afterwards the critical kernels were implemented using a SPP on a FPGA and the speedup from this hardware acceleration was measured. These results from [VS02] were used as the starting point of the case study. It was assumed that the application, which was introduced in Section 3.1, consists of five benchmarks from the MediaBench suite combined into one software application. Each benchmark has one critical kernel and therefore the entire software application has five kernels, each with a possibility of being executed in software or using a SPP. The hardware acceleration results obtained from [VS02] were then used to determine the tradeoffs that exist between executing this software application using the various design alternatives of the current study.

The system architectures of the design alternatives, which were illustrated in Figure 3-1, were already described in Chapter 3. When performing the case study it was also necessary to further specify the system components such as the GPP, memory components and the IC Technology of the FPGA. Overall, it was assumed that the system architecture is similar to the one studied in [VS02]. It is assumed that the GPP is a 32 bit MIPS processor that executes at 100 MHz. The FPGA technology is assumed to be SRAM based with a similar architecture to the Xilinx Virtex II [Xil06a]. Also, as mentioned previously the main memory is a random access memory (RAM) and the non-volatile memory is a flash memory. The FPGA configurations are stored in a read only memory (ROM). It is also assumed that the FPGA can be reconfigured without any help from the GPP. Most commercial FPGAs such as the Xilinx Virtex II have this ability by allowing the FPGA to act as the master on the configuration bus [Xil06a]. It was also assumed that the GPP writes to a register in the FPGA to set which kernel's configuration to use for reconfiguration and to initiate the reconfiguration process of the FPGA. The FPGA was also assumed to have a register that is read by the GPP to determine which kernel's configuration is currently configured in the FPGA. This chapter examines the difference in the overall execution times of the various design alternatives. In Chapter 5, other design criteria such as power consumption, energy consumption, area requirements and unit cost are considered and the tradeoffs between the design alternatives are analyzed. Business and marketing considerations such as TTM and development cost are also considered.

To determine the overall execution times for the different design alternatives it is required to obtain values for the terms that are needed to solve the execution time (t_{total}) equations derived in Chapter 3. In order to use realistic HW acceleration times, the speedup values from [VS02] were used, but the actual benchmarks were not run in the simulation. For the other terms the various configuration scheduling algorithms were first implemented in C/C++. The values for the time delays such as $t_{initiate}$ and t_{check}

were determined by compiling the configuration scheduling algorithms and determining the average instruction counts using the MIPS instruction set simulator that comes as part of the MIPS SDE from [Mip06]. Using the average clock cycles per instruction (CPI) for the 32 bit MIPS processor, which was determined in [VS02] to be 1.5, the time delays were obtained by multiplying the CPI with the average instruction counts.

To estimate the reconfiguration time ($t_{config_{CHW_n}}$) a different approach was required. It was first noted that $t_{config_{CHW_n}}$ is only a factor for design alternatives that use dynamic reconfiguration. It was also noted that no partial reconfiguration is possible in the study since the hardware designs of the SPPs are considered to be independent. Also the FPGA for the dynamically reconfigurable design alternatives has to be large enough such that it can hold the configuration of the critical kernel with the largest SPP hardware design. First, in order to determine the size of the FPGA, the equivalent gate estimation technique described in [FM03] was used to calculate the equivalent gates of the FPGA from the VHDL lines of code for the hardware design of the largest SPP. Then, using ratios from [Bar05], [Xil06b], [Xil06c] and [Sau00] $t_{config_{CHW_n}}$ was obtained. First, the configuration file bit stream size was determined from the equivalent gates of the FPGA using the equivalent gate to configuration file bit stream size ratio obtained from [Sau00] and [Bar05]. Then, the configuration time was calculated by considering how long it would take to configure the FPGA with this configuration file size using the technique described in [Bar05] and [Xil06b]. The estimated values of the timing delays and parameters used in the study are listed in Appendix A.2. The resulting values were determined to be reasonable and in the range of expected values.

Probabilities such as $P_{CHW_n\ not\ cfg}$ and $P_{CHW_n\ reconfig}$ in the execution time equations also need to be determined before numerical results can be obtained for the overall execution time of the design alternatives. These probabilities are only present in the execution time equations of design alternatives that use dynamic reconfiguration and configuration scheduling algorithms. To determine actual values for these probabilities a simulation model in SystemC [Ope06] was developed and the different configuration scheduling algorithms and design alternatives were simulated to evaluate the configuration scheduling algorithm performance.

4.2 Simulation Model

A simulation model in SystemC was developed to compare and measure the performance of the design alternatives. In this section, the SystemC modeling framework is introduced and the implementation of the model is explained as follows: first, the software architecture of the simulation model is introduced, then it is explained how the software application's behavior was modeled and afterwards it is shown how the configuration scheduling algorithms were implemented.

4.2.1 SystemC Modeling Framework

VHDL and Verilog based simulators are used often to simulate and design digital hardware. These simulators are useful for modeling at the behavioral level or Register Transfer Level (RTL) but they are not suited for system level modeling. SystemC was developed to fill the requirement of a true system level modeling language that is suited for fast design space exploration at the preliminary and conceptual design phases. SystemC is implemented as a C/C++ class library and is ideal for hardware/software co-design since it provides language constructs that allow for fast modeling of hardware, software and hardware/software interfaces. SystemC is also advantageous since it is possible to model accurately only the parts of the system, which are of interest. Parts of the system, which are of less interest, can be modeled at a higher abstraction level. This provides fast development time and simulation speed since only important aspects of the system are modeled accurately. SystemC is used often in hardware/software co-design and is especially suited for analyzing hardware/software partitioning problems. For these reasons, SystemC was chosen as the modeling framework for studying the performance of the configuration scheduling algorithms. SystemC allowed the model to be developed while focusing on the parts of the system which are of interest such as the implementation details of the configuration scheduling algorithms, while the rest of the system was modeled at a higher abstraction level.

4.2.2 Model Architecture

A timed functional model was implemented to determine the performance of the various configuration scheduling algorithms. Timed functional models are used often in hardware/software partitioning since they are useful for analyzing performance tradeoffs between different design alternatives [GL02]. The timing delays used by the functional model were estimated as described in Section 4.1. The main goal for developing the simulation model was to determine actual values for the probabilities in the execution time (t_{total}) equations and overall the aim was to test how the various configuration scheduling algorithms

and their implementations perform for different behaviors of the software application. The conceptual software architecture of the simulation model is shown in Figure 4-1 and the modules, subsystems and their interaction is described thereafter.

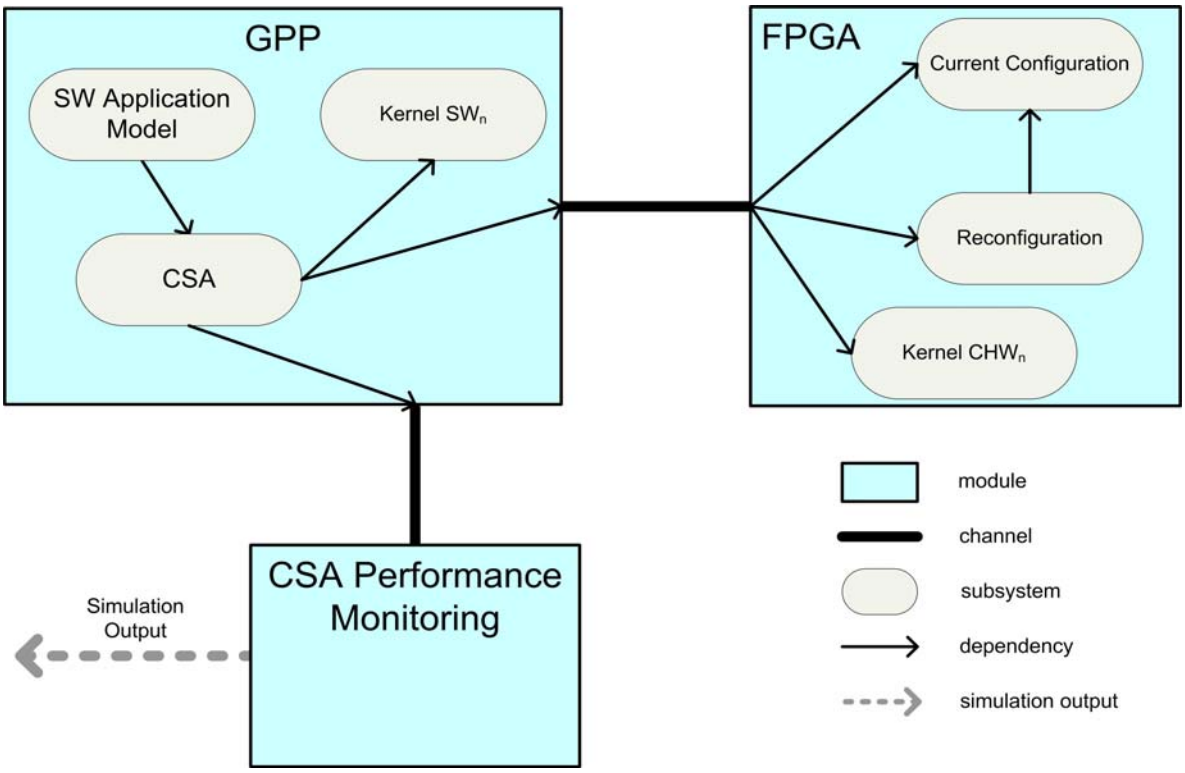

Figure 4-1: SystemC Model Software Architecture

GPP Module: This module simulates all the computation that takes place on the GPP. It consists of the software Application Model, configuration scheduling algorithm and Kernel SW_n subsystems. The software Application Model subsystem is a model of the software application, which simulates key behaviors such as the relationship between critical kernels that affects when and how critical kernels execute. This subsystem simulates the execution of non-kernel software and determines if a critical kernel is currently needed to be executed on the SPP. When one of the critical kernels is required to be executed, it passes control to the configuration scheduling algorithm subsystem, which decides whether to execute the critical kernel in software or using the SPP. If the critical kernel should be executed in software, control is passed to the Kernel SW_n subsystem, which simply executes the critical kernel in software and returns control. Otherwise the configuration scheduling algorithm subsystem communicates with the

FPGA module to execute the critical kernel using the SPP. The configuration scheduling algorithm also communicates with the FPGA module when it is required to initiate the reconfiguration of the FPGA with a different kernel's SPP or when it is required to determine which kernel's SPP is currently configured in the FPGA. In SystemC, communication between modules takes place via channels and in the study, the channels were implemented as FIFO buffers to model the bus interface between the GPP and the FPGA. The software Application Model subsystem and the configuration scheduling algorithm subsystem are explained further in Section 4.2.3 and Section 4.2.4 respectively.

FPGA Module: This module simulates the FPGA and it consists of the Current Configuration, Reconfiguration and Kernel CHW_n subsystems. The Reconfiguration subsystem is used by the GPP module to initiate the reconfiguration of the FPGA with a different kernel's SPP. Once reconfiguration is completed, the Reconfiguration subsystem informs the Current Configuration subsystem about which kernel's SPP is currently configured in the FPGA. The Current Configuration subsystem is then used by the GPP module to determine which kernel's SPP is currently configured in the FPGA. The Kernel CHW_n subsystem is used by the GPP module to execute a critical kernel using its SPP.

Configuration Scheduling Algorithm Performance Monitoring Module: This module obtains the necessary information from the GPP Module's configuration scheduling algorithm subsystem to measure the performance of the configuration scheduling algorithm. Information such as how many times the FPGA is required to be reconfigured and how many times critical kernels are executed using their SPP implementation is obtained to determine actual values for the probabilities in the execution time equations. These probabilities allow the performance of the configuration scheduling algorithms to be compared. The module obtains the simulation results and prints a summary to a console and writes other details to an output file.

The overall software architecture of the simulation model was designed to be to be modular to allow future reuse of the model. The configuration scheduling algorithm and software application behavior have been modularized into separate subsystems to allow different configuration scheduling algorithms and software application models to be easily integrated into the model. The modularity of the simulation model's software architecture also allows it to be easily extended and refined.

4.2.3 Application Models

As mentioned previously, it is assumed that the software application, which was introduced in Section 3.1, consists of five benchmarks from the MediaBench suite. Each benchmark has one critical kernel and

therefore, the entire software application has five kernels, each with a possibility of being executed in software or using a SPP. The software Application Model subsystem is a high level model of this software application. It models and simulates the application's behavior such as the relationship between critical kernels that affects when and how critical kernels execute. A number of different software application behavior models were used in the experiments to test the performance of the different configuration scheduling algorithms. The different application models make up the test cases and are the inputs to the simulation model. Each of the application models has a number of operational modes, through which the application moves during its execution. The models can be grouped into software applications with kernels that have a temporal locality characteristic or a kernel correlation characteristic. The temporal locality behavior of the applications is specified by the probabilities of each critical kernel executing next given that a critical kernel is required by the application. For applications with a temporal locality behavior these probabilities depend on the operational mode. For applications with a kernel correlation behavior these probabilities also depend on which critical kernel executed last. The probabilities can vary over different operational modes of the application, but are assumed to be constant throughout the entire execution time of a single operational mode. For applications with a temporal locality behavior each operational mode has one dominant kernel that executes frequently while the other kernels execute less frequently. It is more crucial to accelerate the dominant kernel of each operational mode than to accelerate the non-dominant kernels. Applications with a kernel correlation behavior have the following behavior. After a kernel's execution, one dominant kernel has a high probability of executing next while the other kernels have a smaller probability of executing. Table 4-1 summarizes and describes the different temporal locality and kernel correlation application models used as the test cases of the study. The statistics describing the different models of the software application's behavior are listed in the tables of Appendix A.1.

Table 4-1: Software Application Models Summary

Test Case	Description	Table
Temporal Locality Test Cases		
1	Only one kernel executes in each operational mode, while the other kernels have a zero probability of executing.	Table A-1
2	In each operational mode one dominant kernel executes with a high probability while the other kernels execute with a lower probability.	Table A-2
3	This test case is similar to temporal locality test case 2 except that non-dominant kernels have a higher probability of executing during each operational mode.	Table A-3
Kernel Correlation Test Cases		
1	After a kernel's execution, only one particular kernel can execute next while the other kernels have a zero probability of executing next.	Table A-4
2	After a kernel's execution, one dominant kernel has a high probability of executing next while the other kernels can have a smaller probability of executing.	Table A-5
3	This test case is similar to kernel correlation test case 2 except that non dominant kernels have a higher probability of executing next.	Table A-6

4.2.4 Configuration Scheduling Algorithm Implementation

The configuration scheduling algorithm subsystem consists of the various configuration scheduling algorithm implementations. The on demand configuration scheduling algorithm was simply implemented as shown previously in Algorithm 3-1. Both the temporal locality and kernel correlation configuration scheduling algorithms were introduced previously in Algorithm 3-2 and Algorithm 3-3, respectively. In this section, further details about the implementation of the monitoring and selection phases of the temporal locality and kernel correlation configuration scheduling algorithms are explained. In Chapter 6 other possible ways to implement the temporal locality and kernel correlation configuration scheduling algorithms will be described and their tradeoffs will be evaluated. To gather information about the software application's temporal locality behavior during the monitoring phase, the temporal locality configuration scheduling algorithm uses a circular buffer to store the history of which kernels executed most frequently in the recent past. The history buffer length is the length of the circular buffer, which indicates the maximum number of entries the buffer can hold. The history buffer data structure and the **update_history_tl_csa** function that is used to update the history buffer during the monitoring phase are shown in Algorithm 4-1.

```
/* history_buffer_length is the length of the circular buffer*/
history[history_buffer_length] //array with history_buffer_length entries
cbuff_index //index to the current location in the circular buffer

/*kernel_required is the kernel the software application is requesting to
execute*/
update_history_tl_csa(kernel_required)
{
    cbuff_index = cbuff_index +1
    if(cbuff_index is equal to  history_buffer_length)
    {
        cbuff_index =0
    }
    history[cbuff_index]=kernel_required
}
```

Algorithm 4-1: Temporal Locality Monitoring Phase

During the selection phase the temporal locality configuration scheduling algorithm simply looks at the entries in the history buffer to determine which kernel executed most frequently in the recent past. The FPGA is then reconfigured with this kernel's configuration if it is not already configured in the FPGA. When determining the critical kernel that executed most frequently in the recent past, sometimes there will be a tie according to the entries in the history buffer. If one of the kernels in the tie has its configuration currently configured in the FPGA, the configuration scheduling algorithm selects this kernel to have the least amount of reconfiguration. If neither of the kernels in the tie has its configuration currently configured in the FPGA, the configuration scheduling algorithm randomly selects one of these kernel's configurations for the reconfiguration of the FPGA.

Similar to the temporal locality configuration scheduling algorithm, the kernel correlation configuration scheduling algorithm uses circular buffers to gather information about a software application's kernel correlation behavior during the monitoring phase. The kernel correlation configuration scheduling algorithm uses multiple circular buffers to store the history of which kernels executed most frequently after each kernel. The history buffer data structures and the **update_history_kc_csa** function that is used to update the history buffers during the monitoring phase are shown in Algorithm 4-2.

```
/* history_buffer_length is the length of each circular buffer*/
/*number_of_kernels are the total number of kernels in the application*/
history[number_of_kernels][ history_buffer_length] //array of circular buffers
cbuff_index [number_of_kernels] //array of circular buffer indices
previous_kernel // is the kernel that executed last

/*kernel_required is the kernel the software application is requesting to
execute*/
update_history_kc_csa(kernel_required)
{
    cbuff_index [previous_kernel]= cbuff_index [previous_kernel]+1
    if(cbuff_index [previous_kernel] is equal to history_buffer_length)
    {
        cbuff_index [previous_kernel]=0
    }
    history [previous_kernel] [cbuff_index [previous_kernel] ]=kernel_required
    previous_kernel=kernel_required
}
```

Algorithm 4-2: Kernel Correlation Monitoring Phase

The kernel correlation configuration scheduling algorithm uses one separate history buffer with the same history buffer length for each critical kernel since the likelihood of a critical kernel executing next depends on which critical kernel executed last. During the selection phase, the kernel correlation configuration scheduling algorithm simply looks at the entries in the history buffer of the currently required kernel to determine which kernel executed most frequently in the recent past after the currently required kernel. The FPGA is then reconfigured with this kernel's configuration if it is not already configured in the FPGA.

4.3 Experiments, Results and Discussion

In this section, the experiments and the simulation results which were performed using the software application models introduced in Section 4.2.3 are presented. The inputs to the simulation model are listed in Appendix A. The statistics describing the different models of the software application's behavior are

listed in Appendix A.1 and are summarized in Table 4-1. Each of the software application models consists of a number of operational modes through which the application moves during its execution. During the simulation the application iterates through the operational modes 10,000 times. It was determined experimentally that the results converge by iterating through the operational modes this many times. A random number generator in C/C++ was used and was seeded differently for each simulation run. It is assumed that the application stays in each operational mode for an equal period of time. The average number of times critical kernels execute during an operational mode before the application moves to a different mode is defined as $Exec_{per\_mode}$. Initially estimated values of the timing delays and parameters (listed in Appendix A.2) were used for the simulations. Some of these values were varied to analyze the effects they have on the simulation results. All of the simulation results are presented in Appendix B.

The overall goal of the experiments was to identify if the simulation results are consistent with the analysis and theory that was presented in Chapter 3. The experiments and the analysis of the results are separated into two different parts. The first part analyzes to which extent the temporal locality configuration scheduling algorithm can reduce and adjust the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. Special attention is given to the effects the history buffer length has on the frequency of reconfiguration. The second part examines the differences in the overall execution time of software applications that results from the use of the different configuration scheduling algorithms and design alternatives. An analysis of the effects the reconfiguration time overhead has on the overall execution time is also presented.

4.3.1 Frequency of Reconfiguration and History Buffer Length Analysis

The goal of this part of the analysis is to study the effects the temporal locality configuration scheduling algorithm has on the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. Experiments were also performed to study the effects of the history buffer length has on the frequency of reconfiguration. The simulation results for these experiments are listed in Appendices B.1 and B.2. First, to illustrate how the history buffer entries of the temporal locality configuration scheduling algorithm change during the execution of a software application, the temporal locality software application test case 1 is used. This test case has the property of only one kernel executing in each operational mode, while the other kernels have a zero probability of executing. The contents of the temporal locality configuration scheduling algorithm's history buffer at the point when this software application changes from mode 1 to mode 2 are shown in Table 4-2.

Table 4-2: Temporal Locality Configuration Scheduling Algorithm History Buffer Example

	Point in Time				
	Time 1	**Time 2**	**Time 3**	**Time 4**	**Time 5**
Mode	Mode 1	Mode 2	Mode 2	Mode 2	Mode 2
history[0]	Kernel 2	Kernel 2	Kernel 2	Kernel 3	Kernel 3
history[1]	Kernel 2	Kernel 2	Kernel 2	Kernel 2	Kernel 3
history[2]	Kernel 2	Kernel 3	Kernel 3	Kernel 3	Kernel 3
history[3]	Kernel 2	Kernel 2	Kernel 3	Kernel 3	Kernel 3
cbuff_index	1	2	3	0	1
kernel_required	2	3	3	3	3
winner	2	2	2	3	3
SPP in FPGA	Kernel 2	Kernel 2	Kernel 2	Reconfigure	Kernel 3

In this example a history buffer length of 4 is used. The software application changes from mode 1 to mode 2 between time 1 and time 2. Only kernel 2 executes in mode 1 and therefore at time 1, the history buffer is initially full with entries of kernel 2 and the FPGA is also configured with the SPP of kernel 2. Afterwards only kernel 3 executes when the operational mode changes to mode 2. At time 2, kernel 3 is required for the first time and one entry of kernel 3 is placed into the circular history buffer. When a kernel is required whose SPP is not configured in the FPGA the temporal locality configuration scheduling algorithm goes into the selection phase to determine if the required kernel's SPP should be configured into the FPGA. Kernel 2 is still selected as the winner during the selection phase at time 2 since the history buffer has more entries of kernel 2 than kernel 3. Therefore the FPGA stays configured with the SPP of kernel 2. At time 3, kernel 3 is required again and again kernel 2 is selected as the winner during the selection phase since the history buffer has two entries of kernel 2 and two entries of kernel 3. As mentioned previously, in case of a tie, the temporal locality configuration scheduling algorithm selects the kernel that is currently configured in the FPGA to have the least amount of reconfiguration. Therefore the FPGA still stays configured with the SPP of kernel 2. At time 4 when kernel 3 is required again, kernel 3 is selected as the winner during the selection phase since the history buffer has more entries of kernel 3 than kernel 2. Therefore the temporal locality configuration scheduling algorithm initiates the reconfiguration of the FPGA with the SPP of kernel 3. At time 5 when kernel 3 is required again, the FPGA is already configured with the SPP of kernel 3 and therefore it can be executed using its SPP. As long as the application stays in mode 2, only kernel 3 will be required by the application and since its SPP is already configured in the FPGA it will be executed using its SPP. This example shows how the entries in the history buffer are used by the temporal locality configuration scheduling algorithm to determine the

most frequent kernel that executes during each mode. Using this information, the FPGA is configured with the kernel's SPP that has the highest likelihood of executing in the near future.

When simulating temporal locality test case 1 using a history buffer length of 6 and an $Exec_{per\ mode}$ of 40, the results in Table B-1 and Table B-6 are obtained for the on demand configuration scheduling algorithm and temporal locality configuration scheduling algorithm. When comparing t_{total} of the CHW_{od_system} and CHW_{tl_system} design alternatives, it can be seen that the CHW_{od_system} design alternative has a shorter execution time than the CHW_{tl_system} design alternative. This is due to the behavior of temporal locality software application test case 1 itself. Only one kernel executes in each operational mode and the on demand configuration scheduling algorithm simply reconfigures the FPGA each time the mode changes and a new kernel's SPP is required. Afterwards, no reconfiguration is required until the next mode change occurs. Every time the mode changes, the CHW_{tl_system} design alternative takes a longer time to reconfigure the FPGA with the SPP of the new required kernel since the history buffer must first be filled with enough entries of this kernel. Until reconfiguration is determined to be required during the selection phase of the temporal locality configuration scheduling algorithm, the kernel is executed in software which accounts for the $P_{CHW_n\ not\ cfg_{TLCSA}}$ of 10%. A higher history buffer length would therefore have a negative affect on the execution time since it will take longer to fill the history buffer with enough entries to invoke reconfiguration. The frequency of reconfiguration of both the CHW_{od_system} and CHW_{tl_system} design alternatives are the same since the reconfiguration only occurs once per mode change in both cases. It can be seen that is more advantageous to use the on demand configuration scheduling algorithm instead of the temporal locality configuration scheduling algorithm for applications where only one kernel executes in each operational mode such as the temporal locality software application test case 1. Due to this, temporal locality test case 1 is excluded from the rest of the study when comparing the temporal locality configuration scheduling algorithm and on demand configuration scheduling algorithm.

When simulating the temporal locality software application test cases 2 and 3 using a history buffer length of 6 and a $Exec_{per\ mode}$ of 40, the results in Table B-2 and Table B-3 are obtained for the on demand configuration scheduling algorithm and temporal locality configuration scheduling algorithm. These software applications have a behavior where a different kernel executes very frequently during each mode, but the other kernels can execute with a lower frequency of occurrence. It can be seen that in both cases, $P_{CHW_n\ not\ cfg_{ODCSA}}$ is larger than $P_{CHW_n\ not\ cfg_{TLCSA}}$ for all kernels. This means that for the CHW_{tl_system} design alternative, the critical kernel's SPPs are often already configured in the FPGA when they are

required. It can also be seen that the CHW_{tl_system} has a significantly lower frequency of reconfiguration than the CHW_{od_system} design alternative. This is due to the fact that the on demand configuration scheduling algorithm has no alternative than to reconfigure the FPGA whenever the previously executed critical kernel differs from the current one. The temporal locality configuration scheduling algorithm, on the other hand, has the ability to execute the kernels in software and only decides to reconfigure the FPGA when the history buffer indicates that the currently required kernel will be executing frequently in the near future. Reducing the frequency of reconfiguration can have positive benefits such as reduced power consumption of the system.

Next, the effects of the history buffer size on the performance and frequency of reconfiguration of the temporal locality configuration scheduling algorithm is examined For this, the temporal locality software application test cases 2 and 3 were simulated while the history buffer length was varied and the $Exec_{per\_mode}$ was kept constant at 40. The graphs in Figure 4-2 and Figure 4-3 illustrate how the average frequency of reconfiguration for all kernels is affected by the history buffer length.

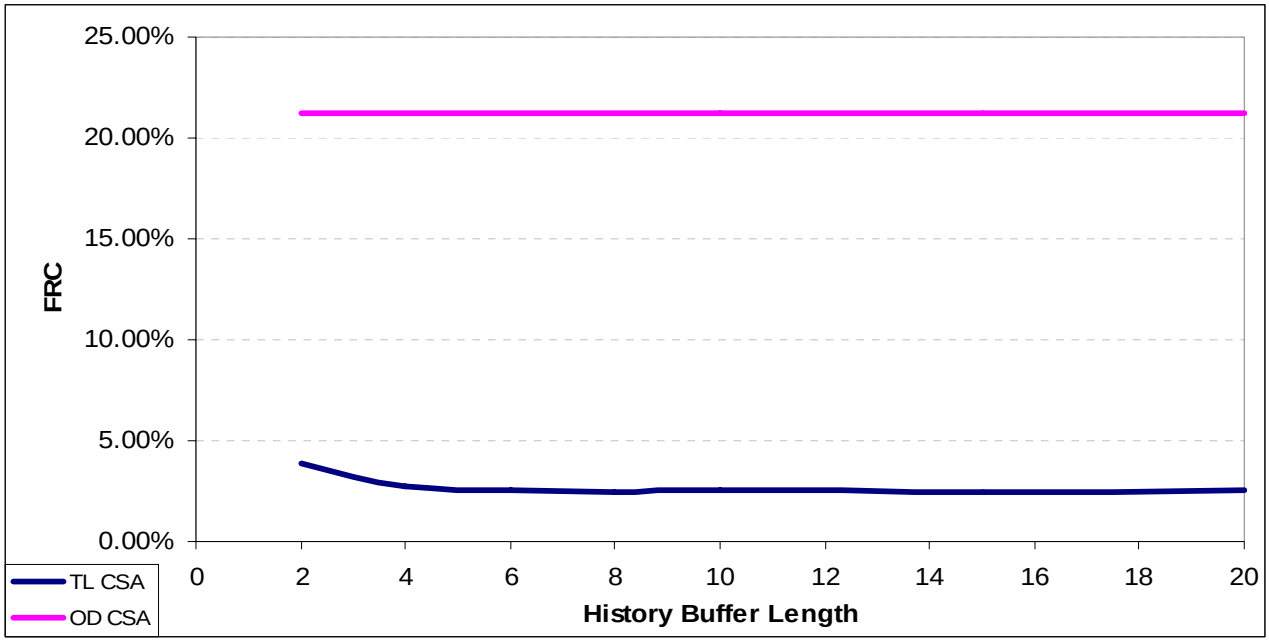

Figure 4-2: Temporal Locality Test Case 2 - Frequency of Reconfiguration vs. Buffer Length

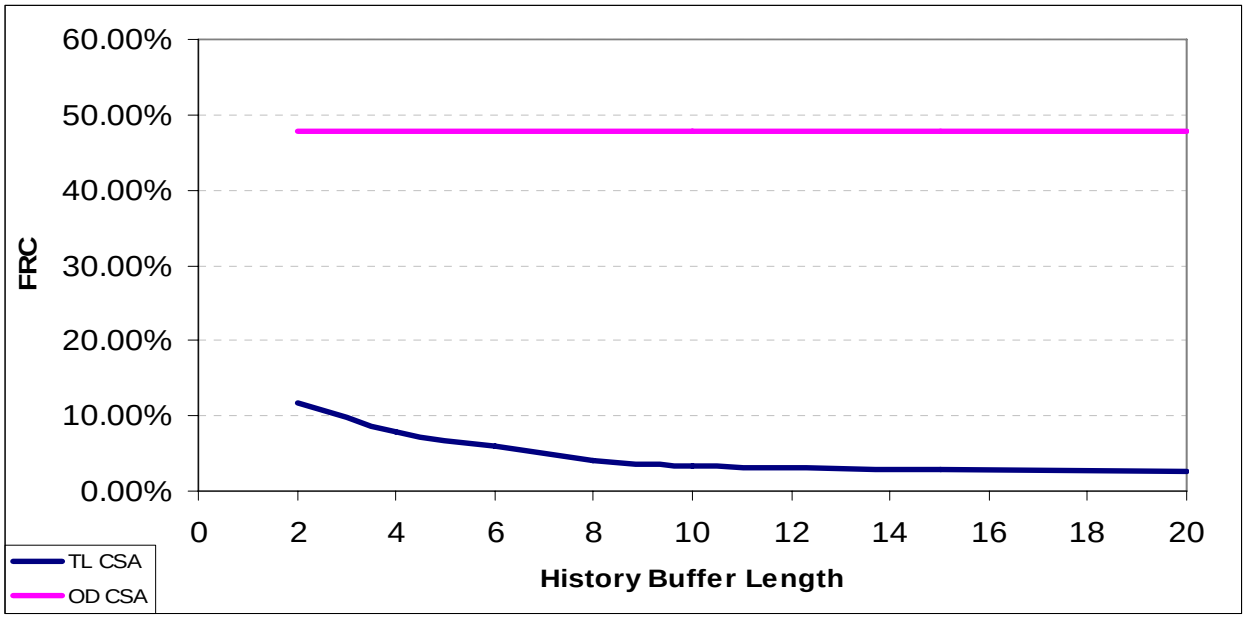

Figure 4-3: Temporal Locality Test Case 3 - Frequency of Reconfiguration vs. Buffer Length

As stated previously it can be seen that both for both test cases, the frequency of reconfiguration of the temporal locality configuration scheduling algorithm is significantly lower than for the on demand configuration scheduling algorithm. As the history buffer length increases, the frequency of reconfiguration of the temporal locality configuration scheduling algorithm decreases. It can also be observed that at large history buffer lengths, there is no significant reduction in the frequency of reconfiguration by any further increase to the history buffer length. The reduction in the frequency of reconfiguration as a result of history buffer length increase is expected since a larger history buffer holds more entries. More entries mean that the temporal locality configuration scheduling algorithm has more memory of which kernels executed in the near past. The temporal locality configuration scheduling algorithm reconfigures the FPGA with a new configuration whenever the kernel which has the most frequent entries in the history buffer changes from one kernel to another. A history buffer with a larger length requires more time to go from having most frequent entries of one kernel to having most frequent entries of another kernel. Due to this the frequency of reconfiguration reduces as the history buffer length increases. A larger history buffer causes the temporal locality configuration scheduling algorithm to look at more long term trends when determining which kernel executed most frequently in the recent past than a shorter history buffer.

The history buffer length also affects the frequency of how often kernels' SPPs are already configured in the FPGA when they are required. This frequency is defined as the frequency of already configured (FAC) and is equivalent to $1 - P_{CHW_n \text{ not } cfg}$. The effect the history buffer length has on the average frequency of already configured of all kernels is illustrated in the graphs in Figure 4-4 and Figure 4-5. Again the same test cases were simulated while the history buffer length was varied and the $Exec_{per\_mode}$ was kept constant at 40. This time, the effects on the frequency of already configured are measured instead of the frequency of reconfiguration.

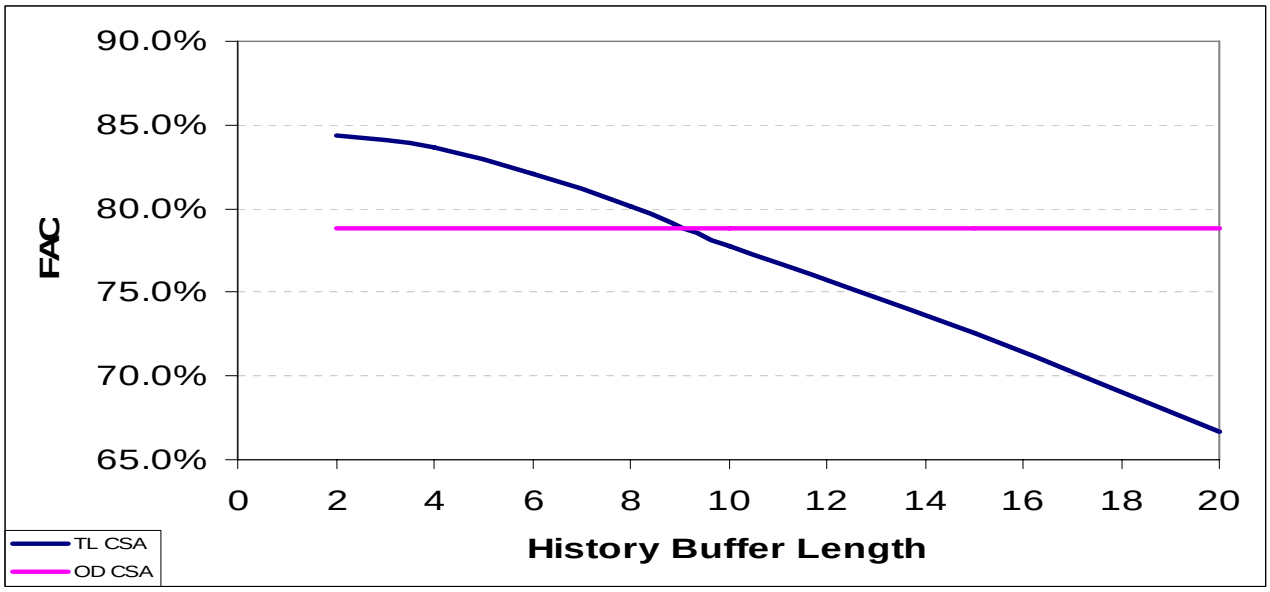

Figure 4-4: Temporal Locality Test Case 2 - Frequency of Already Configured vs. Buffer Length

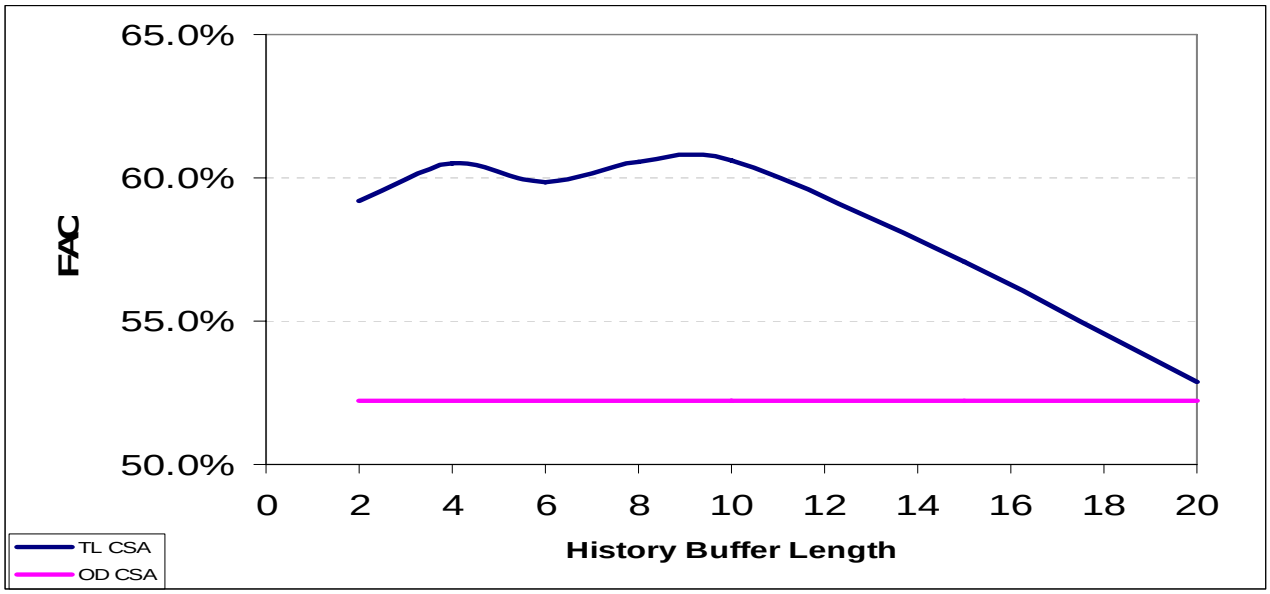

Figure 4-5: Temporal Locality Test Case 3 - Frequency of Already Configured vs. Buffer Length

It is desirable to increase the frequency of already configured to increase the frequency of executing critical kernels using their SPP implementation. By doing so, the overall execution time of the software application is reduced. In contrast to the on demand configuration scheduling algorithm, the frequency of already configured for the temporal locality configuration scheduling algorithm is not only a function of the software application's behavior. Here, the frequency of already configured also depends on the accuracy with which the temporal locality configuration scheduling algorithm implementation predicts the configuration that will be required most frequently in the near future. This accuracy of the temporal locality configuration scheduling algorithm also depends to some degree on the length of the history buffer as shown in the graphs in Figure 4-4 and Figure 4-5 It can be seen that for both test cases when the history buffer length is small, the frequency of already configured of the temporal locality configuration scheduling algorithm is higher than the frequency of already configured of the on demand configuration scheduling algorithm. The temporal locality configuration scheduling algorithm decreases with increasing history buffer length. Eventually at some point the temporal locality configuration scheduling algorithm's frequency of already configured falls below the frequency of already configured of the on demand configuration scheduling algorithm. This is due to the fact that the history buffer length should be large enough to contain enough entries to make accurate predictions while it should be small enough to quickly recognize operational mode changes. If the history buffer length is too small it might not contain enough entries about which kernels executed in the near past, to make accurate predictions about which kernel

will execute most frequently in the near future. If the history buffer length is too large, it might take too long for entries from previous operational modes to be replaced by entries of the current operational mode. Depending on the $Exec_{per\_mode}$ which is the average number of times critical kernels that execute during an operational mode before the application moves to a different mode, the history buffer length might be too long for the temporal locality configuration scheduling algorithm to recognize that a mode change occurred and that a different kernel should be configured into the FPGA for this operational mode. In this experiment, an $Exec_{per\_mode}$ of 40 was used. A history buffer length of 20 requires that kernels execute 20 times after the operational mode changes before the history buffer is full of entries form the current operational mode. It therefore takes too long for accurate predictions to be made by the temporal locality configuration scheduling algorithm about which kernel to configure into the FPGA.

For the temporal locality test case 2 a history buffer length of 2 is sufficiently long for the temporal locality configuration scheduling algorithm to make accurate predictions as shown in Figure 4-4. Any additional increase in the history buffer length reduces the frequency of already configured to a point where the temporal locality configuration scheduling algorithm's frequency of already configured becomes lower than the on demand configuration scheduling algorithm's frequency of already configured. For temporal locality test case 3 a history buffer length of 3 is also sufficiently long for the temporal locality configuration scheduling algorithm to make accurate predictions as shown in Figure 4-5. But an increase in the history buffer length to 4 increases the frequency of already configured, which means that the accuracy of the temporal locality configuration scheduling algorithm increases. Temporal locality test case 3 has a high frequency of already configured when compared to the frequency of already configured of the on demand configuration scheduling algorithm up to a history buffer length of 10. Afterwards any additional increase in the history buffer length reduces the frequency of already configured.

Overall, the frequency of already configured for both the on demand configuration scheduling algorithm and temporal locality configuration scheduling algorithm are lower for temporal locality test case 3 than for temporal locality test case 2. Also the frequency of reconfiguration for both the on demand configuration scheduling algorithm and temporal locality configuration scheduling algorithm are higher for temporal locality test case 3 than for temporal locality test case 2. This is due to the differences in the temporal locality behavior between the two test cases. Temporal locality test case 3 has a higher probability of non dominant kernels executing during each operational mode. This means that there is a high frequency of different critical kernels being executed after another, which results in a higher

frequency of reconfiguration and lower frequency of already configured for both the on demand and temporal locality configuration scheduling algorithm. This difference in the temporal locality behavior between the two test cases accounts for the slight differences in the graphs.

Using these experiments, it was observed that for the temporal locality configuration scheduling algorithm, increasing the history buffer length has the positive benefit of decreasing the frequency of reconfiguration while also having the negative side effect of decreasing the frequency of already configured when the history buffer is increased beyond a certain point. The history buffer length should be chosen such that both the temporal locality configuration scheduling algorithm's frequency of already configured stays sufficiently high, while also providing an acceptably low frequency of reconfiguration. The history buffer length should be large enough to contain enough entries to make accurate predictions while it should also be small enough to quickly recognize operational mode changes. It was observed that the temporal locality configuration scheduling algorithm can have a significantly lower frequency of reconfiguration than the on demand configuration scheduling algorithm and it was illustrated that the frequency of reconfiguration can be adjusted by varying the history buffer length. It was also seen that it is more advantageous to use the on demand configuration scheduling algorithm instead of the temporal locality configuration scheduling algorithm for applications where only one kernel executes in each operational mode such as temporal locality test case 1.

4.3.2 Execution Time Analysis

The goal of this part of the analysis is to study the difference in the overall execution time between the various configuration scheduling algorithms and design alternatives. First, the difference in the execution time between the CHW_{tl_system} and CHW_{od_system} design alternative is studied. Afterwards, the difference in execution time between the CHW_{kc_system} and CHW_{od_system} design alternative is examined. The execution times of the SW_{system} and CHW_{stc_system} design alternatives are also shown to allow for comparison. The analysis also focuses on the effects the reconfiguration time overhead has on the software application execution time. The simulation results for these experiments are presented in Appendix B.3.

To examine the difference in execution time between the CHW_{tl_system} and CHW_{od_system} design alternative and to determine the effects of the reconfiguration time, temporal locality test cases 2 and 3 were simulated while the reconfiguration time was varied. The history buffer length was kept constant at 6 since it was determined to provide a good performance from the previous analyzes. $Exec_{per\_mode}$ was kept constant at 40. The graphs in Figure 4-6 and Figure 4-7 illustrate the simulation results.

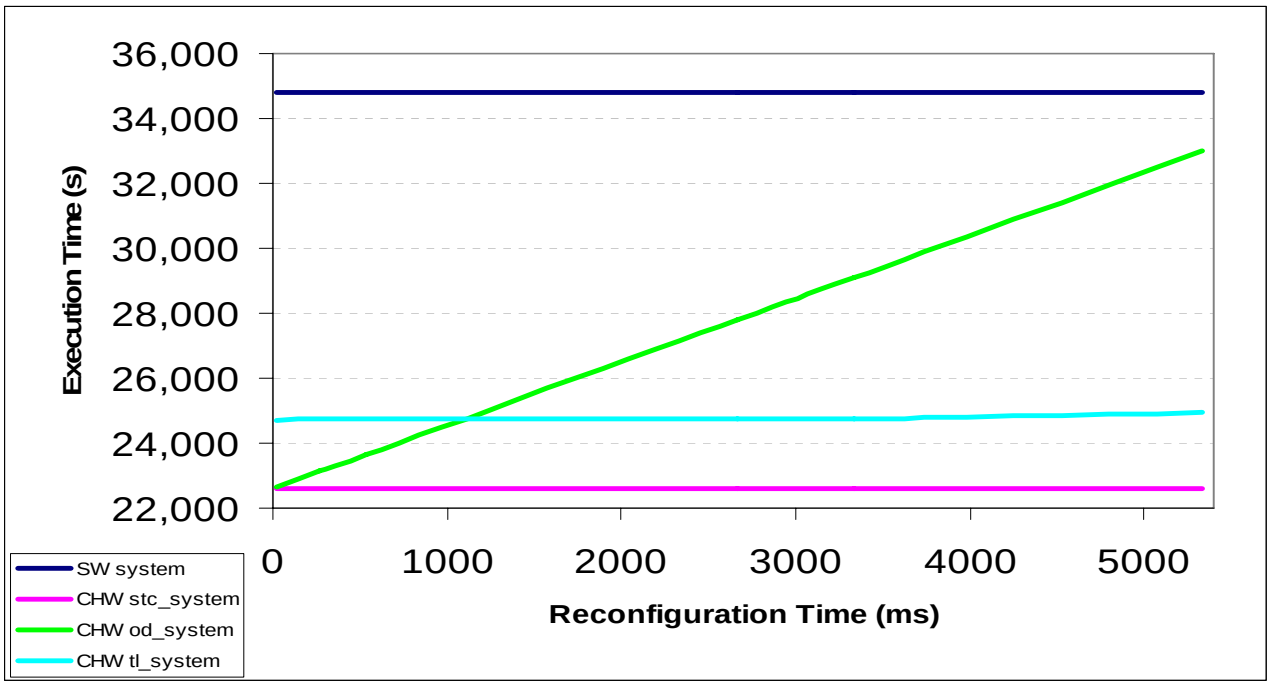

Figure 4-6: Temporal Locality Test Case 2 – Execution Time vs. Reconfiguration Time

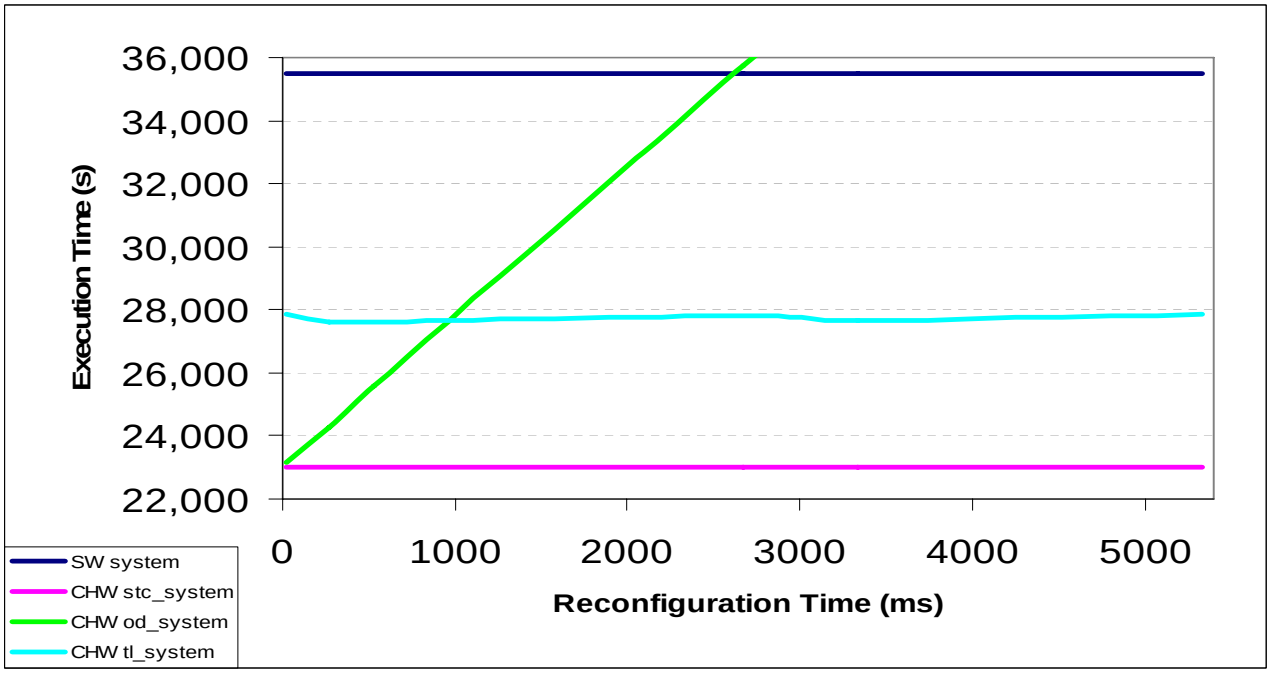

Figure 4-7: Temporal Locality Test Case 3 – Execution Time vs. Reconfiguration Time

The execution time of the SW_{system} design alternative is the time that the other design alternatives try to speed up through hardware acceleration. The CHW_{stc_system} design alternative provides the shortest execution time since the SPPs of the kernels are always configured in the FPGA and therefore, the kernels are always executed using their SPP without any additional overhead such as reconfiguration time. From the graphs in Figure 4-6 and Figure 4-7 it can be seen that the execution time of the CHW_{od_system} design alternative depends on the reconfiguration time. The CHW_{od_system} design alternative's execution time is almost as fast as the CHW_{stc_system} design alternative at small reconfiguration times, but as the reconfiguration time increases the execution time increases to the point where it is worse than the execution time of the SW_{system} design alternative which it is trying to speed up. The CHW_{tl_system} design alternative, on the other hand, does not vary significantly with an increase in the reconfiguration time. The reconfiguration time overhead is therefore not a critical factor for this design alternative. It can be seen that the on demand configuration scheduling algorithm is more advantageous than the temporal locality configuration scheduling algorithm at small reconfiguration times, while the temporal locality configuration scheduling algorithm is more advantageous at longer reconfiguration times. This is due to the fact that at smaller reconfiguration times when a kernel is required to be executed and its SPP is not currently configured in the FPGA, it is faster to reconfigure the FPGA with the SPP and execute the kernel using its SPP instead of executing the kernel using its software implementation. In other words, the inequality in Equation 14 should hold true for all critical kernels for the temporal locality configuration scheduling algorithm to be more advantageous over the on demand configuration scheduling algorithm. The greater this inequality is the more advantageous the temporal locality configuration scheduling algorithm becomes over the on demand configuration scheduling algorithm.

$$t_{config\ \ CHW_n} + t_{start\ \ CHW_n} + t_{CHW_n} + t_{finish\ \ CHW_n} > t_{SWn} \qquad \textbf{(14)}$$

By comparing the two graphs in Figure 4-6 and Figure 4-7 it can be seen that the slope of the CHW_{od_system} design alternative curve in Figure 4-7 is steeper than the curve in Figure 4-6. It can therefore be observed that an increase in reconfiguration time has a larger effect on the execution time of the CHW_{od_system} design alternative for temporal locality test case 3 than for temporal locality test case 2. This is due to the differences in the temporal locality behavior of the two test cases. Temporal locality test case 3 has a higher probability of non dominant kernels executing during each operational mode. This means that there is a high frequency of different critical kernels being executed after another which results in a

higher frequency of reconfiguration. Since the FPGA is reconfigured more often, the reconfiguration time overhead becomes more critical for temporal locality test case 3 than for temporal locality test case 2.

Next the difference in execution time between the CHW_{kc_system} and CHW_{od_system} design alternative is analyzed. Similar to the temporal locality configuration scheduling algorithm, the kernel correlation configuration scheduling algorithm uses circular buffers to gather information about a software application's kernel correlation behavior during the monitoring phase. The kernel correlation configuration scheduling algorithm uses multiple circular buffers to store the history of which kernels executed most frequently after each kernel. It uses one separate history buffer for each critical kernel since the likelihood of each of a critical kernel executing next depends on which critical kernel executed last. During the selection phase, the kernel correlation configuration scheduling algorithm simply looks at the entries in the history buffer of the currently required kernel to determine which kernel executed the most frequently in the recent past after the currently required kernel. The FPGA is then reconfigured with this kernel's configuration if it is not already configured in the FPGA. Similar to the temporal locality configuration scheduling algorithm, the length of the history buffers of the kernel correlation configuration scheduling algorithm should be large enough to contain enough entries to make accurate predictions while they should also be small enough to quickly recognize operational mode changes

To examine the difference in execution time between the CHW_{kc_system} and CHW_{od_system} design alternative and to determine the effects of the reconfiguration time, kernel correlation test cases 1, 2 and 3 were simulated while the reconfiguration time was varied. The history buffer length was kept constant at 3 for all test cases except for kernel correlation test case 1. A history buffer length of 1 was determined to be sufficient for kernel correlation configuration scheduling algorithm to make accurate predictions for kernel correlation test case 1. This is due to the test case's property of only one particular kernel executing next after a kernel, while the other kernels have a zero probability of executing next. The $Exec_{per\_mode}$ was kept constant at 40. The graphs in Figure 4-8, Figure 4-9 and Figure 4-10 illustrate the simulation results.

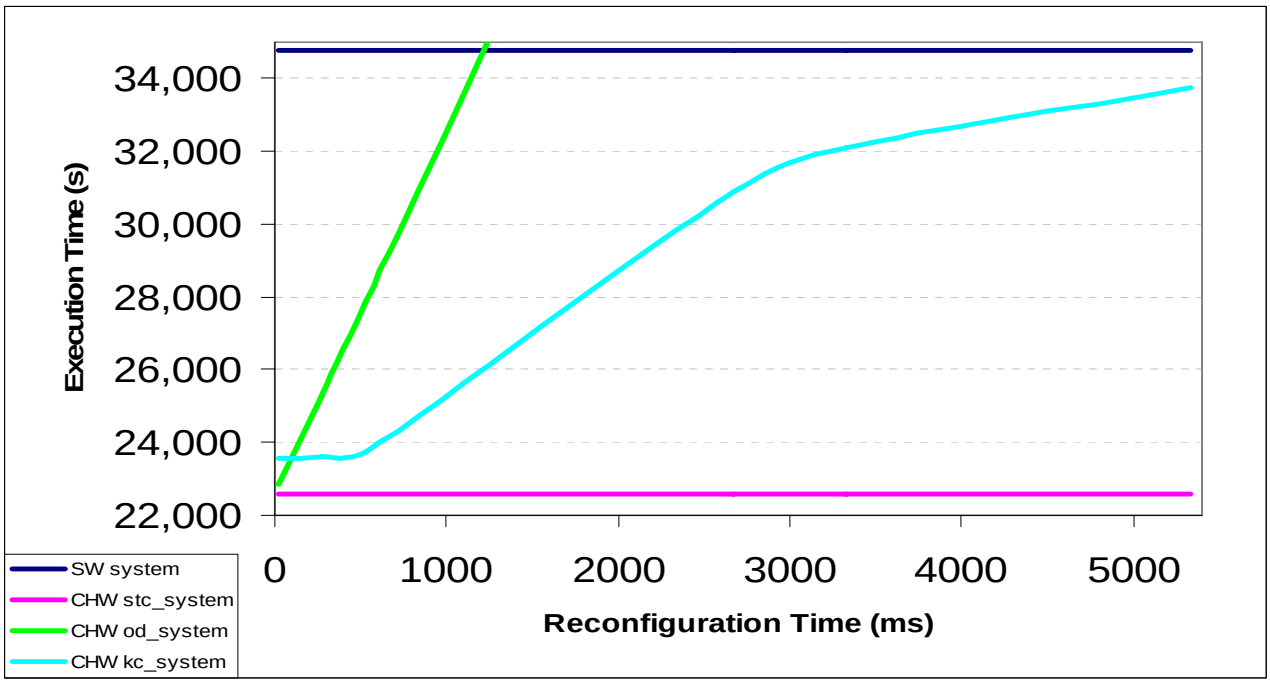

Figure 4-8: Kernel Correlation Test Case 1 – Execution Time vs. Reconfiguration Time

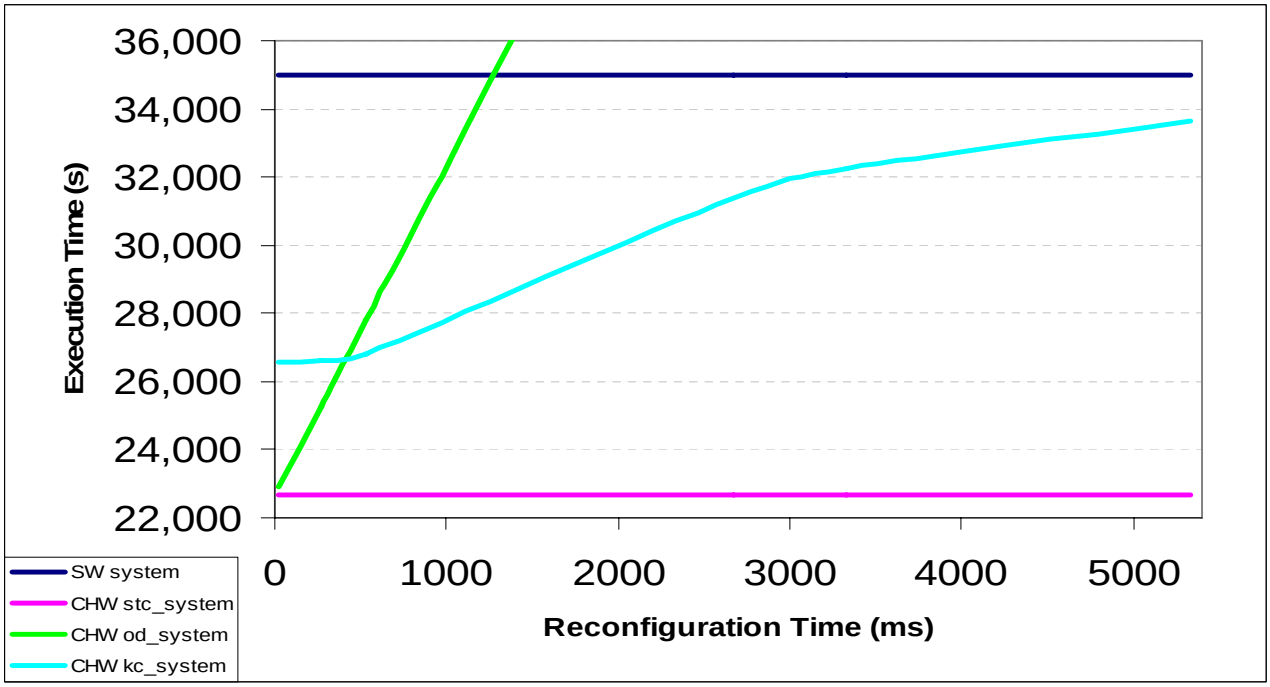

Figure 4-9: Kernel Correlation Test Case 2 – Execution Time vs. Reconfiguration Time

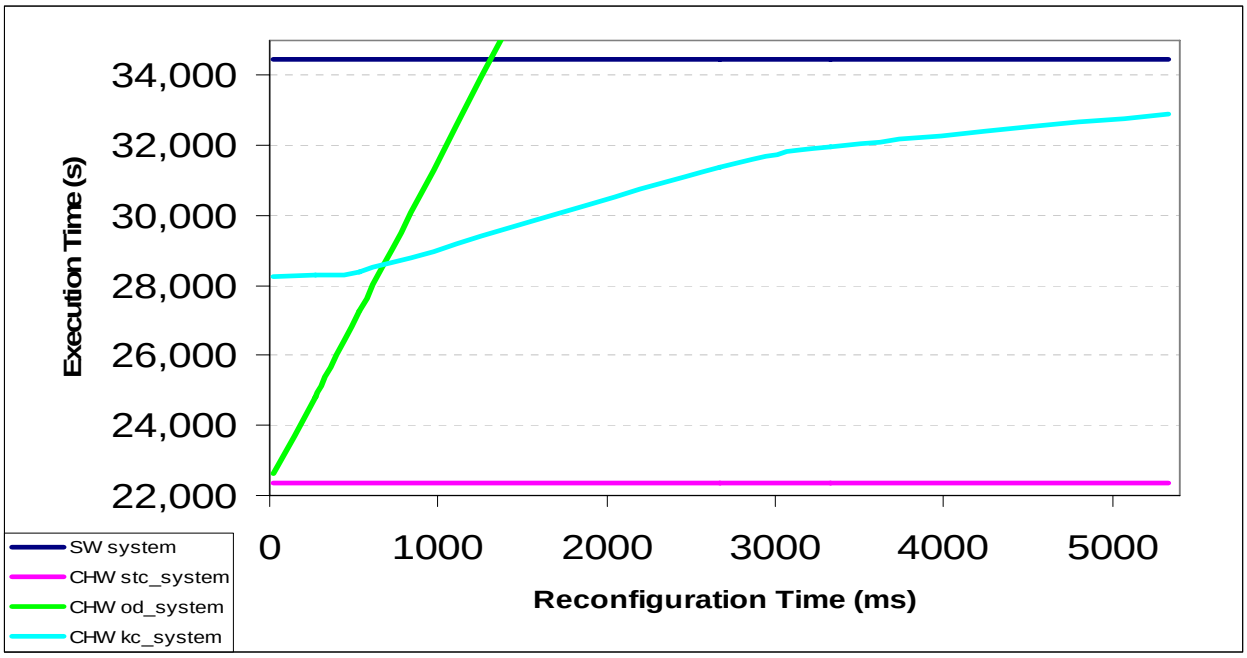

Figure 4-10: Kernel Correlation Test Case 3 – Execution Time vs. Reconfiguration Time

From the graphs in Figure 4-8, Figure 4-9 and Figure 4-10, it can be seen that for all three test cases, the execution times of both the CHW_{od_system} and CHW_{kc_system} design alternatives depend on the reconfiguration time. The CHW_{od_system} design alternative's execution time is almost as fast as the CHW_{stc_system} design alternative at small reconfiguration times, but as the reconfiguration time increases, the execution time increases to the point where the execution time is worse than the execution time of the SW_{system} design alternative. The CHW_{kc_system} design alternative, on the other hand, shows no significant increase in execution time at small reconfiguration times. It can be seen that the on demand configuration scheduling algorithm is more advantageous than the kernel correlation configuration scheduling algorithm at small reconfiguration times. This is due to the fact that at smaller reconfiguration times when a kernel is required to be executed and its SPP is not currently configured in the FPGA, it is faster to reconfigure the FPGA with the SPP and execute the kernel using its SPP instead of executing the kernel using its software implementation. This was already pointed out previously when discussing the temporal locality configuration scheduling algorithm using the inequality in Equation 14.

The kernel correlation configuration scheduling algorithm is therefore more advantageous than the on demand configuration scheduling algorithm at larger reconfiguration times. But after the execution time increases beyond a critical point the execution time of the CHW_{kc_system} design alternative starts to increase

and approaches the execution time of the SW_{system} design alternative at long reconfiguration times. This is due to the fact that the kernel correlation configuration scheduling algorithm is not implemented to reduce the frequency of reconfiguration like the temporal locality configuration scheduling algorithm is. The temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm by only trying to execute the most frequently executing kernel using its SPP while executing the other kernels using their software implementation. The kernel correlation configuration scheduling algorithm and on demand configuration scheduling algorithm on the other hand aim at executing all the kernels using their SPP, which results in a higher frequency of reconfiguration. Each time a kernel is required to execute, the kernel correlation configuration scheduling algorithm looks at the entries in the history buffer of the currently required kernel to determine which kernel executed most frequently in the recent past after the currently required kernel. Since applications with a kernel correlation behavior have a high frequency of different critical kernels being executed after another, the frequency of reconfiguration is high for the kernel correlation configuration scheduling algorithm. Since the FPGA is reconfigured more often, the reconfiguration time overhead becomes more critical for the kernel correlation configuration scheduling algorithm when compared to the temporal locality configuration scheduling algorithm.

The reconfiguration time overhead therefore affects the execution time of the CHW_{kc_system} design alternative since a longer reconfiguration process will make the FPGA unusable for a longer period of time. Even when the kernel correlation configuration scheduling algorithm correctly predicts the kernel that is going to execute next and initiates the reconfiguration of the FPGA with this kernel's SPP, a large reconfiguration time can cause the SPP of this kernel to be not configured in the FPGA on time for it to be used when the kernel is required to execute. At this point the kernel would be executed using its software implementation instead of its SPP implementation causing an increase in the execution time of the software application. Due to this, the execution time of the CHW_{kc_system} design alternative starts to increase and approaches the execution time of the SW_{system} design alternative at long reconfiguration times, since almost all kernels are executed using their software implementation.

The slight difference in the graphs in Figure 4-8, Figure 4-9 and Figure 4-10 are due to the differences in the kernel correlation behavior of the three test cases. Since the slopes of the CHW_{od_system} design alternative curve in all three graphs are similar, it can be observed that an increase in reconfiguration time has a similar effect on the execution time of the CHW_{od_system} design alternative for all three test cases. This is due to the fact that all three test cases have a similar frequency of different critical kernels being

executed after another which results in a similar frequency of reconfiguration. Since the FPGA is reconfigured at a similar frequency, an increase in reconfiguration time has a similar effect on the execution time of the CHW_{od_system} design alternative for all three test cases.

It can also be observed that the execution time of the CHW_{kc_system} design alternative moves closer to the execution time of the SW_{system} design alternative even at short reconfiguration times when going from kernel correlation test case 1 to kernel correlation test case 2 and when going from kernel correlation test case 2 to kernel correlation test case 3. kernel correlation test case 1 has the characteristic that after a kernel's execution, only one particular kernel can execute next, while the other kernels have a zero probability of executing next. Kernel correlation test cases 2 and 3 have the characteristic that after a kernel's execution multiple kernels have the possibility of executing next. One dominant kernel has a high probability of executing next while the other kernels can have a low probability of executing next. Kernel correlation test case 3 has a higher probability of non-dominant kernels executing next than kernel correlation test case 2. Due to this difference, the kernel correlation configuration scheduling algorithm can predict the kernel that is going to execute next more accurately for kernel correlation test case 1 than for the other two test cases. Also the kernel correlation configuration scheduling algorithm makes better predictions for the kernel correlation test case 2 than for the kernel correlation test case 3 since kernel correlation test case 2 has a low probability of non dominant kernels executing next. This is due to the fact that the kernel correlation configuration scheduling algorithm always tries to reconfigure the FPGA with the kernel's SPP that has the highest frequency of executing after the current kernel. The higher the frequency of non-dominant kernels executing next is in the application. the higher the number of incorrect predictions of the kernel correlation configuration scheduling algorithm become. As the number of incorrect predictions increases, more kernels are executed using their software implementation instead of their SPP implementation and the closer the execution time of the CHW_{kc_system} design alternative becomes to the execution time of the SW_{system} design alternative. This is observed in the graphs in Figure 4-8, Figure 4-9 and Figure 4-10 where the execution time of the CHW_{kc_system} design alternative moves closer to the execution time of the of the SW_{system} design alternative even at short reconfiguration times when going from kernel correlation test case 1 which has a zero probability of non-dominant kernels executing next, to kernel correlation test case 3, which has the highest probability of a non-dominant kernel executing next out of the three test cases.

4.4 Summary

This chapter first presented the case study that was used to evaluate the various configuration scheduling algorithms and design alternatives. Afterwards the modeling framework and methodology that was chosen was explained. After introducing the implementation of the model, the experiments and the simulation results were presented. The purpose of the experiments was to examine the differences in the execution time that results from the use of different configuration scheduling algorithms and design alternatives. The experiments also examined the effects the reconfiguration time overhead has on the execution time of the particular software application. Additionally experiments were performed to analyze the extent to which the temporal locality configuration scheduling algorithm can reduce and adjust the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm.

It was observed that for the temporal locality configuration scheduling algorithm, increasing the history buffer length has the positive benefit of decreasing the frequency of reconfiguration, while also having the negative side effect of decreasing the frequency of already configured. It was also observed that the temporal locality configuration scheduling algorithm has a significantly lower frequency of reconfiguration than the on demand configuration scheduling algorithm and it was illustrated that the frequency of reconfiguration can be adjusted by varying the history buffer length. The on demand configuration scheduling algorithm was determined to be more advantageous than the temporal locality configuration scheduling algorithm for applications where only one kernel executes in each operational mode. Overall, for both the temporal locality and kernel correlation configuration scheduling algorithm, it was observed that the history buffer length should be large enough to contain enough entries to make accurate predictions while it should also be small enough to quickly recognize operational mode changes.

It was observed that the execution time of the CHW_{od_system} design alternative depends on the reconfiguration time. The CHW_{tl_system} design alternative on the other hand does not vary significantly with an increase in the reconfiguration time. Overall, the on demand configuration scheduling algorithm is more advantageous than the temporal locality configuration scheduling algorithm at small reconfiguration times, while the temporal locality configuration scheduling algorithm is more advantageous at longer reconfiguration times. The CHW_{kc_system} design alternative also shows no significant increase in execution time at small reconfiguration times. But after the execution time increases beyond a critical point, the execution time of the CHW_{kc_system} design alternative starts to increase and approaches the execution time of the SW_{system} design alternative at long reconfiguration times.

Overall, the on demand configuration scheduling algorithm is more advantageous than the kernel correlation configuration scheduling algorithm at small reconfiguration times, while the kernel correlation configuration scheduling algorithm is more advantageous at larger reconfiguration times.

Chapter 5
Tradeoff Analysis and Design Space Exploration

The CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system} design alternatives are examples of dynamically reconfigurable systems, while the CHW_{stc_system} design alternative is a statically configured system. As mentioned previously the main advantage of dynamically reconfigurable systems is that a single, smaller FPGA which contains multiple configurations over time could be used. Dynamic reconfiguration is useful in systems where only one of the configurations is needed at a time. In contrast, a statically configured system requires multiple FPGAs or a larger FPGA to hold all of the configurations at all times. Dynamically reconfigurable systems therefore can have a higher functional density due to the smaller FPGA which results in reduced unit cost and area. The previous chapters examined the difference in the execution time between the design alternatives. This chapter first examines and estimates other important design criteria such as power consumption, energy consumption, area requirements and unit cost. Afterwards, business and marketing considerations such as time to market (TTM) and development cost are also considered. The goal of this chapter is to provide an overall understanding about how to evaluate when a design alternative is more advantageous by determining the tradeoffs that exist in the design space.

5.1 Estimation

This section explains how the power consumption, energy consumption, area requirements and unit cost of the various design alternatives can be estimated. Through the estimation process, the critical factors affecting when a design alternative is advantageous can be determined. During estimation, differences in area requirements, unit cost, power consumption and energy consumption resulting from the difference in memory requirements and from differences in bus routing and other interconnections between system components are ignored. The differences in memory requirements between the various design alternatives can be ignored since kernels are loops that make up only a small part of all the instructions of a software application. Also the additional instructions and data required for the configuration scheduling algorithms considered in the study are small when compared to the entire memory requirements of a software application. The additional overhead of the ROM that is used to store the FPGA configurations is considered during the estimation since it can have a significant effect on the usefulness of the various design alternatives. Also, the differences between the design alternatives resulting from differences in bus

routing and other interconnections between system components are ignored, since they can be assumed to be negligible when compared to the system components themselves as shown in [Tan06] and [VS02]. The factors that were left out can be easily incorporated into the analysis provided in this section when more accurate estimation is required.

5.1.1 Power and Energy Consumption

This section explains how the power and energy consumption of the various design alternatives can be estimated. Power consumption is an important design criterion when designing embedded systems. It affects important factors such as the heat dissipation of the system. It also affects the battery life by having an impact on the energy consumption of the system. The power consumption can be estimated by adding up the average power of all the components of the system as shown in [VS02]. For each component, the overall total average power consumption can be determined by considering the average power consumption of the component in each mode and by considering the percentage of time the component spends in each mode. This is shown in Equation 15, which is used to estimate the power consumption of the various design alternatives. The parameters and terms used in the equation are explained in Table 5-1.

$$power_{total} = power_{RoS_{total}} + power_{GPP_{total}} + power_{FPGA_{total}} + power_{ROM_{total}} \quad \textbf{(15)}$$

where

$$power_{GPP_{total}} = Pcnt_{GPP_{active}} \times power_{GPP_{active}} + Pcnt_{GPP_{inactive}} \times power_{GPP_{inactive}}$$

$$power_{FPGA_{total}} = Pcnt_{FPGA_{active}} \times power_{FPGA_{active}} + Pcnt_{FPGA_{inactive}} \times power_{FPGA_{inactive}} + Pcnt_{FPGA_{reconfig}} \times power_{FPGA_{reconfig}}$$

$$power_{ROM_{total}} = Pcnt_{ROM_{active}} \times power_{ROM_{active}} + Pcnt_{ROM_{inactive}} \times power_{ROM_{inactive}}$$

Table 5-1: Power Estimation Parameters

Term	Description
$power_{total}$	Total average power consumption of the system
$power_{GPP_{total}}$	Total average power consumption of the GPP.
$power_{FPGA_{total}}$	Total average power consumption of the FPGA.
$power_{ROM_{total}}$	Total average power consumption of the ROM
$power_{RoS_{total}}$	Total average power consumption of the rest of the system which includes everything except the average power of the GPP, FPGA and ROM. This term is independent of the design choice.
$power_{GPP_{active}}$	Average power consumption of the GPP when in active mode. The GPP is always in active mode except during the time when critical kernels are executed using their SPPs on the FPGA. $Pcnt_{GPP_{active}}$ is defined as the percentage of time when the GPP is in active mode.
$power_{GPP_{inactive}}$	Average power consumption of the GPP when in inactive mode. The GPP goes into inactive mode when critical kernels are executed using their SPPs on the FPGA. $Pcnt_{GPP_{inactive}}$ is defined as the percentage of time when the GPP is in inactive mode.
$power_{FPGA_{active}}$	Average power consumption of the FPGA when in active mode. The FPGA goes into active mode when critical kernels are executed using their SPPs on the FPGA. $Pcnt_{FPGA_{active}}$ is defined as the percentage of time when the FPGA is in active mode.
$power_{FPGA_{inactive}}$	Average power consumption of the FPGA when in inactive mode. The FPGA is always in inactive mode except during the time when critical kernels are executed using their SPPs on the FPGA and during the time when the FPGA is reconfigured. $Pcnt_{FPGA_{inactive}}$ is defined as the percentage of time when the FPGA is in inactive mode.
$power_{FPGA_{reconfig}}$	Average power consumption of the FPGA when in reconfiguration mode. The FPGA is in reconfiguration mode when the FPGA is reconfigured with a new configuration. $Pcnt_{FPGA_{reconfig}}$ is defined as the percentage of time when the FPGA is in reconfiguration mode.
$power_{ROM_{active}}$	Average power consumption of the ROM when in active mode. The ROM is in active mode when the reconfiguration of the FPGA occurs. During this time the FPGA configuration information stored in the ROM is used for the reconfiguration of the FPGA. $Pcnt_{ROM_{active}}$ is defined as the percentage of time when the ROM is in active mode.
$power_{ROM_{inactive}}$	Average power consumption of the ROM when in inactive mode. The ROM is always in inactive mode except during the time when the reconfiguration of the FPGA occurs. $Pcnt_{ROM_{active}}$ is defined as the percentage of time when the ROM is in inactive mode.

Power consumption data from [VS02] was used to estimate $power_{GPP_{active}}$ and $power_{GPP_{inactive}}$ for the MIPS processor. $power_{FPGA_{active}}$ and $power_{FPGA_{inactive}}$ were determined using FPGA power consumption

estimation techniques described in [SK02] and [Alt05]. $power_{FPGA_{reconfig}}$ was assumed to be the same as $power_{FPGA_{active}}$ which is a reasonable approximation according to experimental results from [SJ02]. $power_{ROM_{active}}$ and $power_{ROM_{inactive}}$ were determined using ROM power consumption estimation techniques described in [KH06]. $power_{RoS_{total}}$ was assumed to be negligible and considered to be zero for the purpose of focusing only on the differences in power consumption that arise by choosing between the different design alternatives. The percentages of time the components spend in each mode can be determined as shown in the equations in Table 5-2 .

Table 5-2: Component Mode Percentage Equations

Term	Equations	
$Pcnt_{GPP_{inactive}}$	$Pcnt_{GPP_{inactive}} = \frac{t_{HW\ execution}}{t_{total}}$	**(16)**
$Pcnt_{GPP_{active}}$	$Pcnt_{GPP_{active}} = 1 - Pcnt_{GPP_{active}}$	**(17)**
$Pcnt_{FPGA_{active}}$	$Pcnt_{FPGA_{active}} = \frac{t_{HW\ execution}}{t_{total}}$	**(18)**
$Pcnt_{FPGA_{reconfig}}$	$Pcnt_{FPGA_{reconfig}} = \frac{t_{total\_reconfig}}{t_{total}}$	**(19)**
$Pcnt_{FPGA_{inactive}}$	$Pcnt_{FPGA_{inactive}} = 1 - Pcnt_{FPGA_{active}} - Pcnt_{FPGA_{reconfig}}$	**(20)**
$Pcnt_{ROM_{active}}$	$Pcnt_{ROM_{active}} = \frac{t_{total\_reconfig}}{t_{total}}$	**(21)**
$Pcnt_{ROM_{active}}$	$Pcnt_{ROM_{inactive}} = 1 - Pcnt_{ROM_{active}}$	**(22)**

It was assumed that the FPGA is not in active mode during $t_{start_{CHW_n}}$ and $t_{finish_{CHW_n}}$ since the power consumption of writing to the FPGA registers can be assumed to be small. As defined in Chapter 3, t_{total} is the total execution time for the entire software application and $t_{HW\ execution}$ is the total time

when critical kernels are executed using their SPPs on the FPGA. $t_{total\_reconfig}$ is the total time that is used to reconfigure the FPGA at runtime. $t_{HW\ execution}$ and $t_{total\_reconfig}$ can be determined for the different design alternatives using the parameters introduced in Chapter 3 as shown in Table 5-3.

Table 5-3: Power Timing Calculations

System	Equations
SW_{system}	$power_{FPGA_{total}} = 0; \quad power_{ROM_{total}} = 0; \quad t_{HW\ execution} = 0;$ therefore : $power_{total} = power_{RoS_{total}} + power_{GPP_{active}}$ **(23)**
CHW_{stc_system}	$t_{HW\ execution} = \sum_{n=1}^{N} \left[t_{CHW_n} \times NOE_n \right]$ $t_{total\_reconfig} = 0$ **(24)**
CHW_{od_system}	$t_{HW\ execution} = \sum_{n=1}^{N} \left[t_{CHW_n} \times NOE_n \right]$ $t_{total\_reconfig} = \sum_{n=1}^{N} \left[P_{CHW_n\ not\ cfg_{ODCSA}} \times t_{config_{CHW_n}} \times NOE_n \right]$ **(25)**
CHW_{tl_system}	$t_{HW\ execution} = \sum_{n=1}^{N} \left[\left(1 - P_{CHW_n\ not\ cfg_{TLCSA}}\right) \times t_{CHW_n} \times NOE_n \right]$ $t_{total\_reconfig} = \sum_{n=1}^{N} \left[P_{CHW_n\ reconfig_{TLCSA}} \times t_{config_{CHW_n}} \times NOE_n \right]$ **(26)**
CHW_{kc_system}	$t_{HW\ execution} = \sum_{n=1}^{N} \left[\left(1 - P_{CHW_n\ not\ cfg_{KCCSA}}\right) \times t_{CHW_n} \times NOE_n \right]$ $t_{total\_reconfig} =$ $\sum_{n=1}^{N} \left[\left(P_{CHW_n\ reconfig\ cfg_{KCCSA}} + P_{CHW_n\ reconfig\ notcfg_{KCCSA}} \right) \times t_{config_{CHW_n}} \times NOE_n \right]$ **(27)**

From the execution time results obtained in Chapter 4 for the test cases in Appendix A.1, the power consumption results in Table 5-4 and Table 5-5 were obtained.

Table 5-4: Temporal Locality Power and Energy Consumption Results

Term	SW_{system}	CHW_{stc_system}	CHW_{od_system}	CHW_{tl_system}
Temporal Locality Test Case 2				
$power_{total}$	105.0 mW	103.9 mW	104.5 mW	104.3 mW
t_{total}	34,812 s	22,606 s	27,803 s	24,742 s
$energy_{total}$	3.66 kJ	2.35 kJ	2.91 kJ	2.58 kJ
Temporal Locality Test Case 3				
$power_{total}$	105.0 mW	103.9 mW	105.1 mW	104.8 mW
t_{total}	35,471 s	23,014 s	35,763 s	27,465 s
$energy_{total}$	3.72 kJ	2.39 kJ	3.76 kJ	2.88 kJ
Reconfiguration frequency = 10kHz				

Table 5-5: Kernel Correlation Power and Energy Consumption Results

Term	SW_{system}	CHW_{stc_system}	CHW_{od_system}	CHW_{kc_system}
Kernel Correlation Test Case 1				
$power_{total}$	105.0 mW	103.9 mW	104.5 mW	104.5 mW
t_{total}	34,760 s	22,573 s	27,906 s	23,758 s
$energy_{total}$	3.65 kJ	2.34 kJ	2.92 kJ	2.48 kJ
Kernel Correlation Test Case 2				
$power_{total}$	105.0 mW	103.9 mW	104.5 mW	104.9 mW
t_{total}	34,976 s	22,680 s	27,816 s	26,788 s
$energy_{total}$	3.67 kJ	2.36 kJ	2.91 kJ	2.81 kJ
Kernel Correlation Test Case 3				
$power_{total}$	105.0 mW	103.9 mW	104.5 mW	105.2 mW
t_{total}	34,484 s	22,377 s	27,287 s	28,384 s
$energy_{total}$	3.62 kJ	2.32 kJ	2.85 kJ	2.98 kJ
Reconfiguration frequency = 50kHz				

It can be seen that the power consumption of the SW_{system} design alternative is higher than the power consumption of the CHW_{stc_system} design alternative. This is due to the fact that for the CHW_{stc_system} design

alternative, the GPP goes into an inactive mode while the critical kernels are executed using their SPPs. In the inactive mode, the GPP consumes less power. The power consumption of the dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}) are higher than the power consumption of the CHW_{stc_system} design alternative due to the additional power consumption from runtime FPGA reconfiguration. The power consumption of the CHW_{tl_system} design alternative is slightly lower than the power consumption of the CHW_{od_system} design alternative due to the fact that the temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration. By reducing the frequency of reconfiguration, the FPGA is reconfigured less often and therefore the power consumption is reduced. As pointed out in Chapter 4, the temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm, by only trying to execute the most frequently executing kernel using its SPP while executing the other kernels using their software implementation. The kernel correlation configuration scheduling algorithm and on demand configuration scheduling algorithm, on the other hand, aim at executing all the kernels using their SPP which results in a higher frequency of reconfiguration. Due to this, the power consumption of the CHW_{kc_system} design alternative is not lower than power consumption of the CHW_{od_system} design alternative. The power consumption of the CHW_{kc_system} design alternative actually becomes higher than the power consumption of the CHW_{od_system} design alternative for kernel correlation test case 2 and 3. This is due to the fact that for these test cases, the kernel correlation configuration scheduling algorithm makes increasingly more incorrect predictions about which kernel is most likely to execute next as pointed out in Chapter 4. Since the FPGA is less frequently configured with the required SPP, more kernels are executed in software on the GPP which increases the power consumption. Overall it can be observed that there is no significant variation in the power consumption of the various design alternatives.

Energy consumption is another important design criterion when designing embedded systems. It affects important factors such as the battery life of the system. The energy consumption of the system is obtained by multiplying the total power consumption by the total execution time of the software application as shown in Equation 28. For the test cases in Appendix A.1, the energy consumption results in Table 5-4 and Table 5-5 were obtained.

$$energy_{total} = power_{total} \times t_{total} \qquad \textbf{(28)}$$

The SW_{system} design alternative has a long execution time when compared to the other design alternatives since no hardware acceleration is used. The power consumption of the SW_{system} design alternative, on the other hand, is similar to the power consumption of the other design alternatives. This results in high energy consumption for the SW_{system} design alternative for all test cases. Overall, it can be observed that the difference in energy consumption between all the different design alternatives depends significantly on difference in the total execution time of the software application. This is due to the fact that there is only a slight variation in the power consumption of the various design alternatives but there is a significant variation in the total execution time of the software application. Design alternatives with faster execution times also have the additional benefit of providing reduced energy consumption. Therefore, all the discussion in Chapter 4 on the differences in the execution times of the various design alternatives also applies to the energy consumption of the system.

5.1.2 Area Requirements and Unit Cost

When comparing the area requirements between the SW_{system} design alternative and the other design alternatives, it can be observed that the main difference is due to the additional area from the FPGA and the ROM used to store the configuration files. This is shown in Equation 29 which is used to estimate the difference in area between the SW_{system} design alternative and the other design alternatives. The parameters and terms used in the equation are explained in Table 5-6.

$$area_{difference} = area_{FPGA_{total}} + area_{ROM_{total}} \quad \textbf{(29)}$$

Table 5-6: Area Estimation Parameters

Term	Description
$area_{difference}$	This is the difference in area between the SW_{system} design alternative and the other design alternatives.
$area_{FPGA_{total}}$	This is the area of the FPGA. Table 5-7 shows how this term can be estimated for the different design alternatives.
$area_{ROM_{total}}$	This is the area of the ROM that is used to store the configuration files. Table 5-8 shows how this term can be estimated for the different design alternatives.
$area_{FPGA_{CHW_n}}$	This is the FPGA area required for the SPP of kernel n.
$area_{ROM_{cfgn}}$	This is the area of ROM required to store one FPGA configuration file.
$area_{ROM_{CHW_n}}$	This is the area of ROM that would be required to store the configuration for the hardware design of kernel n, if the FPGA is just large enough for the hardware design of kernel n.

Table 5-7: FPGA Area Estimation

System	Equations
CHW_{stc_system}	$area_{FPGA_{total}} = \sum_{n=1}^{N} \left[area_{FPGA_{CHW_n}} \right]$ **(30)**
CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}	$area_{FPGA_{total}} = \text{MAX}\left[area_{FPGA_{CHW_1}}, area_{FPGA_{CHW_2}} \ldots area_{FPGA_{CHW_N}} \right]$ **(31)**

Table 5-8: ROM Area Estimation

System	Equations
CHW_{stc_system}	$area_{ROM_{total}} = area_{ROM_{cfgn}}$ where $area_{ROM_{cfgn}} = \sum_{n=1}^{N} \left[area_{ROM_{CHW_n}} \right]$ **(32)**
CHW_{od_system}, CHW_{tl_system} **and** CHW_{kc_system}	$area_{ROM_{total}} = area_{ROM_{cfgn}} \times N$ where $area_{ROM_{cfgn}} =$ MAX $\left[area_{ROM_{CHW_1}}, area_{ROM_{CHW_2}} \ldots area_{ROM_{CHW_N}} \right]$ **(33)**

$area_{FPGA_{CHW_n}}$ was calculated as follows: using the equivalent gates based area estimation techniques described in [Smi97] for a 90 nm processing technology, the size of a Standard Cell based ASIC was estimated. Then, the ASIC to FPGA size conversion factor from [KR06] was used to obtain the size of the FPGA. $area_{ROM_{CHW_n}}$ was estimated as follows: first the configuration file size was determined from the equivalent gates of the FPGA using the equivalent gate to configuration file bit stream size ratio obtained from [Sau00] and [Bar05]. Then the area per bit of ROM for a 90 nm processing technology which was obtained from [KH06] was used to estimate the area of ROM required for the entire configuration file.

The FPGA area of the CHW_{stc_system} design alternative must be large enough to contain the SPP hardware designs of all the critical kernels. The FPGA area of the dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}), on the other hand, must only be large enough to contain the largest SPP hardware design, since the FPGA is only configured with one SPP at a time. The ROM of the CHW_{stc_system} design alternative only contains one configuration file. The configuration file itself contains all configuration information for all the SPPs. The ROM of the dynamically reconfigurable design alternatives, on the other hand, contains N configurations. Each configuration file is as large as the configuration of the largest SPP hardware design. The $area_{difference}$ estimation results for the design

alternatives are presented in Table 5-9. Table 5-10 shows the relative differences in the sizes of the hardware designs in equivalent gates for the different kernels' SPPs. Two different scenarios were considered to show the effects of the relative differences in the sizes of the hardware designs on the increase in functional density of the dynamically reconfigurable design alternatives. The only difference between the two scenarios is that in Scenario 1, kernel 5 has a SPP with a significantly larger hardware design when compared to the other kernels' SPPs and in Scenario 2, kernel 5 has a SPP with a hardware design that is comparable in size to the other design alternatives.

Table 5-9: Area and Unit Cost Results

Term	CHW_{stc_system}	CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}
Scenario 1		
$area_{difference}$	100,689 μm^2	51,363 μm^2
$cost_{UNIT\ difference}$	15.87 $	11.45 $
Scenario 2		
$area_{difference}$	62,750 μm^2	14,897 μm^2
$cost_{UNIT\ difference}$	9.89 $	3.32 $

Table 5-10: SPP Hardware Design Relative Sizes

	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
Scenario 1	103.7%	100.0%	133.9%	111.9%	461.6%
Scenario 2	103.7%	100.0%	133.9%	111.9%	118.3%

In Scenario 1, the additional area of the dynamically reconfigurable design alternatives is around two times smaller than the additional area of the CHW_{stc_system} design alternative. This results in a slight increase in the functional density of the system. In Scenario 2, the additional area of the dynamically reconfigurable design alternatives is around four times smaller than the additional area of the CHW_{stc_system} design alternative. It was determined that in Scenario 1, the dynamically reconfigurable design alternatives have only a smaller improvement in functional density when compared to the CHW_{stc_system} design alternative due to the fact that kernel 5 has a SPP with a significantly larger hardware design when compared to the other kernels' SPPs. This large hardware design forces the FPGA size of the dynamically reconfigurable design alternatives to be large enough for the SPP with the large hardware design even though the other kernels' SPPs do not require such a large FPGA. This also causes the configuration files

to be large and therefore increases the size of the ROM, which contains the configurations files. Overall it can be observed that it is desirable to have all hardware designs at a similar size to improve the functional density increase through the use of the dynamically reconfigurable design alternatives. It should be noted that the impact $area_{difference}$ has on the overall area of the design alternatives depends on the size of the rest of the system which is equivalent to the area of the SW_{system} design alternative. If the area of the rest of the system is small compared to $area_{difference}$, then the additional area ($area_{difference}$) will have a significant impact when choosing between the various design alternatives.

The unit cost is the recurring cost that is associated with each unit of the product that is sold. When comparing the unit cost of the SW_{system} design alternative and the other design alternatives, it can be observed that the main difference is due to the additional FPGA and the ROM used to store the configuration files. This is shown in Equation 34 which is used to estimate the difference in unit cost between the SW_{system} design alternative and the other design alternatives. The parameters and terms used in the equation are explained in Table 5-11.

$$cost_{UNIT_{difference}} = cost_{PART_{FPGA}} + cost_{PART_{ROM}} + cost_{PRODUCTION_{difference}} \quad \textbf{(34)}$$

Table 5-11: Unit Cost Estimation Parameters

Term	Description
$cost_{UNIT_{difference}}$	The difference in unit cost between the SW_{system} design alternative and the other design alternatives.
$cost_{PART_{FPGA}}$	The part cost of the FPGA
$cost_{PART_{ROM}}$	The part cost of the ROM
$cost_{PRODUCTION_{difference}}$	The difference in the production cost of the system between the SW_{system} design alternative and the other design alternatives. This difference arises due to the additional manufacturing cost of the system as a result of the integration of the FPGA and ROM.

Using the equivalent gates based cost estimation technique described in [Smi97] for a 90 nm processing technology, the part cost of the FPGA and ROM was estimated. The equivalent gates of the FPGA were calculated using the equivalent gate estimation technique described in [FM03]. From the size of the

ROM, the equivalent gates of the ROM were calculated using the ROM bit to gate ratio obtained from [KH06] for a 90 nm processing technology. $cost_{PRODUCTION_{difference}}$ was assumed to be negligible when compared to the part costs and was considered to be zero to concentrate on the part costs.

The unit cost estimation results for the two scenarios are presented in Table 5-9. The part costs of the FPGA and ROM are proportional to the size of the FPGA and ROM respectively. Due to this, it is also beneficial to decrease the area of the FPGA and ROM to decrease the unit cost. In Scenario 1, the additional unit cost of the dynamically reconfigurable design alternatives is around 1.4 times smaller than the additional unit cost of the CHW_{stc_system} design alternative. In Scenario 2, the additional unit cost of the dynamically reconfigurable design alternatives is around three times smaller than the additional unit cost of the CHW_{stc_system} design alternative. This is again due to the fact that in Scenario 1, kernel 5 has a SPP with a significantly larger hardware design when compared to the other kernels' SPPs and in Scenario 2, kernel 5 has a SPP with a hardware design that is comparable in size to the other design alternatives. Overall, it can be seen that since the part costs of the FPGA and ROM are proportional to their respective sizes, it is also desirable to have all hardware designs at a similar size to increase the unit cost reduction while also improving the functional density increase through the use of the dynamically reconfigurable design alternatives.

5.2 Business and Marketing Considerations

When designing an actual system it is also important to consider business and marketing design criteria to obtain a competitive solution. Design criteria such as execution speed, power consumption, energy consumption, area requirements and unit cost, which were considered so far, can have a direct impact on the number of units of the product that will be sold. A smaller product with a faster execution speed and longer battery life sold at the same price as another product for example will be valued more by customers and therefore more units could be sold. The design criteria considered so far can therefore impact business criteria such as the number of units that will be sold. To make an estimate of the impact that the different design alternatives have, it is necessary to perform business studies for the particular product's market. Other business and marketing considerations that should be taken into account when choosing one of the design alternatives for the final implementation of the system are the development cost and TTM that are associated with the design alternatives.

5.2.1 Development Cost

The development cost is the non recurring cost that is independent of the number of units of the product that are sold. It consists of the tool cost and labor cost associated with each of the design alternatives. It can be observed that in the study the SW_{system} design alternative has the lowest development cost since no labor and tool costs associated with the hardware acceleration of the kernels is present. The tool costs associated with the FPGA EDA tools are not present and the extra labor cost needed to implement the kernels' SPPs on the FPGA is not required. For all the other design alternatives this labor and tool cost associated with the hardware acceleration of the kernels is present. The dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}) have another additional labor cost associated with developing the configuration scheduling algorithms and the use of the dynamic reconfiguration functionality of the FPGA. Especially the labor cost associated with the development of the configuration scheduling algorithm for the CHW_{tl_system} and CHW_{kc_system} design alternatives can be high due to the tuning to the software application that is required for the temporal locality and kernel correlation configuration scheduling algorithms. The development and testing of the temporal locality and kernel correlation configuration scheduling algorithms can add a significant amount of additional labor cost for these design alternatives.

5.2.2 Time to Market

The TTM is the time between the initial conceptual design of the system to the time when the first unit is sold in the market. It is often crucial to reduce the TTM to enter the market ahead of competition. A short TTM can result in a higher number of units that are being sold to customers. It can be observed that in the study the SW_{system} design alternative has the shortest TTM since no development time is required for the hardware acceleration of the kernels. All the other design alternatives have a longer TTM since they require extra development time for the hardware acceleration of the kernels. Also the dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}) require additional development time for the design and testing of the configuration scheduling algorithms and the use of the dynamic reconfiguration functionality of the FPGA. This results in an increase in the TTM for these design alternatives. Especially the development time associated with the design of the configuration scheduling algorithm for the CHW_{tl_system} and CHW_{kc_system} design alternatives can be high due to the tuning to the software application that is required for the temporal locality and kernel correlation configuration scheduling algorithms. Therefore the TTM of the CHW_{tl_system} and CHW_{kc_system} design alternatives is the longest from all the design alternatives.

5.3 Summary

After examining the difference in the execution time between the design alternatives in the previous chapters, this chapter considered other important design criteria. The goal of this chapter was to examine how to evaluate when which design alternative would be more advantageous by determining the tradeoffs that exist in the design space. First it was explained how the power consumption, energy consumption, area requirements and unit cost of the various design alternatives can be estimated. Through the estimation process, critical factors and the tradeoffs between the design alternatives were determined. One interesting observation that was made when examining the power consumption of the design alternatives was the fact that the power consumption of the CHW_{tl_system} design alternative is slightly lower than the power consumption of the CHW_{od_system} design alternative due to the fact that the temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. The kernel correlation configuration scheduling algorithm on the other hand does not reduce the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm and therefore the power consumption of the CHW_{kc_system} design alternative is not lower than power consumption of the CHW_{od_system} design alternative. Overall it was observed that there is no significant variation in the power consumption of the various design alternatives. It was also determined that the difference in energy consumption between the design alternatives depends significantly on difference in the total execution time of the software application. This is due to the fact that there is only a slight variation in the power consumption of the various design alternatives but there is a significant variation in the total execution time of the software application. Design alternatives with faster software application execution times therefore also have an additional benefit of providing reduced energy consumption.

The main advantage of the dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}) over the CHW_{stc_system} design alternative is that a single smaller FPGA, which contains multiple configurations over time, can be used. The dynamically reconfigurable design alternatives can therefore have a higher functional density due to the smaller FPGA, which results in reduced area requirements. From the analysis it was observed that it is desirable to have all hardware designs at a similar size to improve the functional density increase through the use of the dynamically reconfigurable design alternatives. Since the part costs of the FPGA and ROM are proportional to their respective sizes it is also desirable to have all hardware designs at a similar size to increase the unit cost reduction through the use of the dynamically reconfigurable design alternatives.

In the study it was observed that the SW_{system} design alternative has the lowest development cost and TTM since no overhead associated with the hardware acceleration of the kernels is present. For all the other design alternatives the development cost and TTM associated with the hardware acceleration of the kernels is present. The dynamically reconfigurable design alternatives have an additional development cost and TTM associated with developing the configuration scheduling algorithms and the use of the dynamic reconfiguration functionality of the FPGA. Especially the development time and TTM associated with the development of the configuration scheduling algorithm for the CHW_{tl_system} and CHW_{kc_system} design alternatives can be high due to the tuning to the software application that is required for the temporal locality and kernel correlation configuration scheduling algorithms. Overall it was observed that each of the design alternatives considered in the study has its own advantageous and disadvantageous over the other design alternatives. Different alternatives can therefore turn out to be more advantageous in different scenarios. It is therefore necessary to estimate the design criteria that are important to the situation using estimation techniques such as the ones proposed in the study. Afterwards the design alternative that is the most suited for the current scenario can be determined.

Chapter 6

Summary and Concluding Remarks

This chapter first summarizes the contributions and observations made in the study. Afterwards suggestions on the direction of future research in the area of the study are provided.

6.1 Thesis Contributions and Conclusions

The study consisted of two parts. First the performance of the various configuration scheduling algorithms was examined and the difference in the execution time between the various design alternatives was studied. Afterwards other design criteria such as power consumption, energy consumption, area requirements and unit cost were analyzed. Business and marketing considerations such as TTM and development cost were also considered. The following sections summarize the contributions made in the study.

6.1.1 Execution Time and Configuration Scheduling Algorithm Performance Analysis

The execution time and configuration scheduling algorithm performance analysis provided the following insights and contributions to the study of dynamic reconfiguration and DHSP.

- It was shown that configuration scheduling algorithms can be used to perform DHSP using statically generated configurations. This approach combines the advantages and eliminates some of the drawbacks of both the on demand configuration scheduling algorithm and the approach studied in [SL03] and [LV04]. DHSP configuration scheduling algorithms eliminate the need for profiling and synthesis tool access at runtime, by using statically generated configurations. DHSP configuration scheduling algorithms can also reduce the reconfiguration time overhead by providing an alternate execution path for critical kernels using their software implementation.
- By eliminating the reconfiguration time overhead the DHSP configuration scheduling algorithm based design alternatives (CHW_{tl_system} and CHW_{kc_system}) can be used in systems where timing constraints cannot depend on the reconfiguration time.
- Using the SystemC based simulation model it was illustrated how the various configuration scheduling algorithms and design alternatives can be evaluated. It was shown how such a simulation environment can be used to examine the effects factors such as the reconfiguration time

overhead and history buffer length have on the execution time of a software application. Overall it was illustrated how such a simulation environment can be used to tune the implementation of the configuration scheduling algorithms to the software application at hand.

- It was observed that the execution time of the CHW_{od_system} design alternative depends on the reconfiguration time. The CHW_{tl_system} design alternative on the other hand does not vary significantly with an increase in the reconfiguration time. Overall the on demand configuration scheduling algorithm is more advantageous than the temporal locality configuration scheduling algorithm for small reconfiguration times, while the temporal locality configuration scheduling algorithm is more advantageous for longer reconfiguration times.

- The CHW_{kc_system} design alternative also shows no significant increase in execution time at small reconfiguration times. But the execution time of the CHW_{kc_system} design alternative starts to increase and approaches the execution time of the SW_{system} design alternative at long reconfiguration times. Overall the on demand configuration scheduling algorithm is more advantageous than the kernel correlation configuration scheduling algorithm at small reconfiguration times, while the kernel correlation configuration scheduling algorithm is more advantageous at larger reconfiguration times.

- It was also shown that the temporal locality configuration scheduling algorithm's ability to execute the critical kernels using software or the SPP can be used to reduce and adjust the frequency of reconfiguration. The temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm, by only trying to execute the most frequently executing kernel using its SPP while executing the other kernels using their software implementation. The kernel correlation configuration scheduling algorithm and on demand configuration scheduling algorithm on the other hand aim at executing all the kernels using their SPP, which results in a higher frequency of reconfiguration. For the test cases examined in the study it was observed that the temporal locality configuration scheduling algorithm has a significantly lower frequency of reconfiguration than the on demand configuration scheduling algorithm and it was illustrated that the frequency of reconfiguration can be adjusted by varying the history buffer length.

- It was determined that the on demand configuration scheduling algorithm is more advantageous than the temporal locality configuration scheduling algorithm for applications where only one kernel executes in each operational mode.

- For both the temporal locality and kernel correlation configuration scheduling algorithm it was shown that the history buffer length should be large enough to contain enough entries to make accurate predictions while it should also be small enough to quickly recognize operational mode changes.
- Overall it was shown that the DHSP based configuration scheduling algorithms provide the flexibility of executing critical kernels using their software and SPP implementation. This flexibility allows DHSP based configuration scheduling algorithms to be designed to perform smart scheduling of FPGA configurations and it allows tuning of the configuration scheduling algorithm to the requirements of the application. This was not possible with the on demand configuration scheduling algorithms, which does not possess this flexibility.

6.1.2 Tradeoff Analysis and Design Space Exploration

In the second part of the study design criteria such as power consumption, energy consumption, area requirements and unit cost were analyzed. Business and marketing considerations such as TTM and development cost are also considered. Through this study the overall tradeoffs that exist in the design space were examined. This analysis provided the following insights and contributions to the study of dynamic reconfiguration and DHSP.

- It was observed that the power consumption of the CHW_{tl_system} design alternative is slightly lower than the power consumption of the CHW_{od_system} design alternative due to the fact that the temporal locality configuration scheduling algorithm reduces the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm. The kernel correlation configuration scheduling algorithm on the other hand does not reduce the frequency of reconfiguration when compared to the on demand configuration scheduling algorithm and therefore the power consumption of the CHW_{kc_system} design alternative is not lower than power consumption of the CHW_{od_system} design alternative.
- It was also determined that the difference in energy consumption between the design alternatives depends significantly on difference in the total execution time of the software application. This is due to the fact that there is only a slight variation in the power consumption of the various design alternatives but there is a significant variation in the total execution time of the software application. Design alternatives with faster software application execution times therefore also have an additional benefit of providing reduced energy consumption.

- The main advantage of the dynamically reconfigurable design alternatives (CHW_{od_system}, CHW_{tl_system} and CHW_{kc_system}) over the CHW_{stc_system} design alternative is that a single smaller FPGA, which contains multiple configurations over time, can be used. The dynamically reconfigurable design alternatives can therefore have a higher functional density due to the smaller FPGA, which results in reduced area requirements. From the analysis it was observed that it is desirable to have all hardware designs at a similar size to improve the functional density increase through the use of the dynamically reconfigurable design alternatives. Since the part costs of the FPGA and ROM are proportional to their respective sizes it is also desirable to have all hardware designs at a similar size to increase the unit cost reduction through the use of the dynamically reconfigurable design alternatives.
- The study showed that the SW_{system} design alternative has the lowest development cost and TTM since no overhead associated with the hardware acceleration of the kernels is present. For all the other design alternatives the development cost and TTM associated with the hardware acceleration of the kernels is present. The dynamically reconfigurable design alternatives have an additional development cost and TTM associated with developing the configuration scheduling algorithms and the use of the dynamic reconfiguration functionality of the FPGA. Especially the development time and TTM associated with the development of the configuration scheduling algorithm for the CHW_{tl_system} and CHW_{kc_system} design alternatives can be high due to the tuning of the temporal locality and kernel correlation configuration scheduling algorithms to the software application at hand.

6.2 Remarks on Further Studies

This section provides remarks on further studies and lists guidelines to follow when designing a configuration scheduling algorithm.

6.2.1 Future Research

This section discusses the potential future research in the area of the study. The analysis of configuration scheduling algorithms and DHSP using precompiled configurations can be extended in the following ways.

- In the study the temporal locality and kernel correlation configuration scheduling algorithms were implemented using a circular history buffer that holds the information of which kernels executed in

the past. During the selection phase the configuration scheduling algorithms selected the kernel that has the most number of entries in the history buffer. The study could be extended by looking at other techniques that can be used to implement the temporal locality and kernel correlation configuration scheduling algorithms. One such technique could be to use timestamps associated with when kernels executed in the past. Using the timestamps, kernels that executed further in the past could be weighted less than kernels that executed in the closer to the present when selecting which kernel's configuration to configure into the FPGA.

- It would be beneficial to do more validation of the implementation of the temporal locality and kernel correlation configuration scheduling algorithms by performing further case studies. Also the study examined so far DHSP based configuration scheduling algorithms for software applications with temporal locality and kernel correlation characteristics. The study could be extended by looking at other types of software characteristics and by designing DHSP based configuration scheduling algorithms for software applications with such characteristics.
- It would also be valuable to consider implementing a general DHSP based configuration scheduling algorithm that can detect the characteristic of the software application out of a set of software application types at runtime in order to dynamically adapt and select the best scheduling algorithm for the software application at hand. This kind of configuration scheduling algorithm would be able to handle a wide variety of software application types and can be used in general purpose processing applications such as PCs. The configuration scheduling algorithm could be provided as part of the Operating System and the software application's developers would not be required to design the configuration scheduling algorithm and would not be required to know about the implementation details of the configuration scheduling algorithm. This is due to the fact that the tuning of configuration scheduling algorithms to the needs of each software application would not be required since the general configuration scheduling algorithm would be able to detect the characteristic of the software application at runtime to dynamically adapt and select the best scheduling algorithm for the software application at hand.
- The study looked at dynamic reconfiguration using loosely coupled FPGAs that act as coprocessors. Future studies could extend the study to also consider the use of DHSP configuration scheduling algorithms for GPPs with tightly coupled configurable functional units (FUs). These functional units are more suited to perform fine grain hardware acceleration at the instruction level instead of accelerating entire critical kernels.

- The study can also be extended by looking at how configuration scheduling algorithms can be designed for multi-processor systems with multiple GPPs and a single dynamically reconfigurable FPGA. Also systems with multiple dynamically reconfigurable FPGAs each with the capability of holding the configuration of a kernel's SPP could be examined.
- For the estimation techniques described in Section 5.1, the differences in area requirements, unit cost, power consumption and energy consumption resulting from the difference in memory requirements and from differences in bus routing and other interconnections between system components were ignored. Future work could incorporate these and other factors into the existing estimation models. Through this the estimation models would be refined and the differences between the design alternatives could be compared more accurately.
- The current study did not examine the impact of the variation in the software execution times. The application model could be extended to include variations in the software execution times. A future study could examine whether other configuration scheduling algorithms could be developed to target applications with large variances in software execution time.

6.2.2 Configuration Scheduling Algorithm Design Guidelines

This section provides a guideline on issues that should be considered when configuration scheduling algorithms are designed.

- During the monitoring phase of the configuration scheduling algorithm, information about the software application's behavior, which is useful in the selection phase, is gathered. During the monitoring phase only dynamically changing properties of the software application's behavior should be monitored. Information that can be obtained statically and that is useful for the selection phase should be simply acquired at design time.
- For the temporal locality and kernel correlation configuration scheduling algorithm implementation considered in the study the monitoring, selection and reconfiguration phases are invoked together each time when a critical kernel is required to execute. These phases could also be invoked separately during other stages of the software application's execution. Whenever any information that could be useful in the selection phase can be obtained during the execution of the software the configuration scheduling algorithm could be updated with this information by invoking the

monitoring phase. Whenever it might be beneficial to reconfigure the FPGA with the configuration of another kernel's SPP the selection and reconfiguration phases could be invoked.

- The execution time of the selection and monitoring phases should be kept small to reduce the overhead of executing the configuration scheduling algorithm. A fast and simple configuration scheduling algorithm on the other hand might make too many incorrect predictions, which increases the execution time of the software application. When designing a configuration scheduling algorithm the developer is faced with a tradeoff between the speed and accuracy of the configuration scheduling algorithm.
- During the selection and monitoring phase of the temporal locality and kernel correlation configuration scheduling algorithms computation for tracking kernel executions over a window of limited size is performed. A tradeoff also exists depending on whether the selection or monitoring phase does most the computation. It is possible to move some of the computation in the current implementation of the selection phase to the monitoring phase by calculating which kernel executed most frequently in the monitoring phase. Since the selection phase is not executed each time a kernel needs to run, for the temporal locality configuration scheduling algorithm, it is expected that leaving most the computation in the selection phase is more beneficial on the overall execution time. The kernel correlation configuration scheduling algorithm on the other hand executes the selection phase each time a kernel needs to run and therefore moving more computation into the update phase might be beneficial on the overall execution time. Future studies could be performed to explore this tradeoff further.
- The selection phase should be designed such that all configurations are selected to be configured in the FPGA at some point in time during the execution of the software application. If a configuration is never selected it becomes useless and the overhead of considering it as a possible candidate for the reconfiguration of the FPGA during the selection phase becomes unnecessary.

6.3 Final Observations

The tradeoffs between the currently existing approach for dynamic reconfiguration (on demand configuration scheduling algorithm) and the DHSP configuration scheduling algorithm based approach proposed in the study were evaluated using a case study. It was illustrated how different types of DHSP configuration scheduling algorithms can be implemented and how their performance can be evaluated using a variety of software applications. It was also shown how to evaluate when which of the approaches

would be more advantageous by determining the tradeoffs that exist between them. The underlying factors that affect when which design alternative is more advantageous were also determined and analyzed. The study showed that configuration scheduling algorithms, which perform DHSP using statically generated configurations, can be developed to combine the advantages and reduce some major disadvantages of current approaches. The situations where DHSP configuration scheduling algorithms can be more advantageous than the other approaches were analyzed and illustrated.

Bibliography

[Alt05] Altera Corporation, San Jose, California. *Stratix II vs. Virtex-4 Power Comparison & Estimation Accuracy Whitepaper*, 2005.

[Bar05] Rod Barto, NASA Office of Logic Design. Reconfigurable Computing: Current Status and Potential for Spacecraft Computing Systems. In *Proceedings of the 2005 MAPLD International Conference*, September 2005.

[BC05] A.C. Beck and L. Carro. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *Proceedings of the 42nd Annual Conference on Design Automation*, pages 732-737, 2005.

[BD94] M. Bolotski, A. DeHon and T. Knight. Unifying FPGAs and SIMD Array. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, 1994.

[BH07] B. Boehm, E. Horrowitz. *COCOMO 2.0 Model Definition manual*, version 1.2, 1997.

[Bis03] William D. Bishop. Configurable Computing for Mainstream Software Applications. PhD Thesis, University of Waterloo, 2003.

[Bla05] David C. Black, Jack Donovan. *SystemC: From the Ground Up*. Springer, 2005.

[Con05] J. Cong, Instruction Set Extension with Shadow Registers for Configurable Processors. In *Proceedings of the ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 99-106, 2005.

[Coo02] H. Cook. *A Simple Model for Demand and Product Management*, 2002.

[Cri05] CriticalBlue. *Boosting Software Processing Performance with Coprocessor Synthesis Whitepaper*, June, 2005.

[CW01] H. Cook and A. Wu. On the Valuation of Goods and Selection of the best Design Alternative. In *Proceedings of the 2001 Research in Engineering Design*, volume 13, pages 42-54, 2001.

[DM97] James A. Debardelaben, Vijay K. Madisetti, Anthony J. Gadient. Incorporating Cost Modeling in Embedded-System Design. In *Proceedings of the IEEE Des. Test,* volume 14, issue 3, pages 24-35, 1997.

[Fen04] Peter A. Fenyes, General Motors R&D and Planning, *Multidisciplinary Analysis and Optimization of Vehicle Architectures*, 2004.

[FM03] W.Fornaciari, P.Micheli, F.Salice and L.Zampella. A First Step Towards Hw/Sw Partitioning of UML Specifications. In *Proceedings of the 2003 Conference on Design, Automation and Test in Europe*, page 10668, 2003.

[FS01] W. Fornaciari, F. Salice, U. Bondi and E. Magini. Development Cost and Size Estimation Starting from High-Level Specifications. In *Proceedings of the Ninth international Symposium on Hardware/Software Codesign*, pages 86-91, 2001.

[GB05] Marc Geilen, Twan Basten, Bart Theelen and Ralph Otten. An Algebra of Pareto Points. In *Proceedings of Fifth International Conference on Application of Concurrency to System Design*, pages 88-97, 2005.

[GH05] S. Gatzka, and C. Hochberger. The AMIDAR Class of Reconfigurable Processors. In *the Journal of Supercomputing*, volume 32, issue 2, pages 163-181, May 2005.

[GL02] Thorsten Grötker, Stan Liao, Grant Martin, Stuart Swan. *System Design with SystemC*. Springer, 2002.

[Har95] W. Hardt. An automated approach to HW/SW-codesign. In *Proceedings of the 1995 Partitioning in Hardware-Software Codesign IEE Colloquium*, pages 4/1-1/11, February 1995.

[HM01] W. Huang and E. McCluskey. Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 137-146, 2001.

[HT04a] Ben Hounsell and Richard Taylor, CriticalBlue, *Accelerating software using co-processor synthesis*, 2004.

[HT04b] Ben Hounsell and Richard Taylor, CriticalBlue. Co-processor Synthesis: A New Methodology for Embedded Software Acceleration. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 682-683, 2004.

[HW97] J. Hauser and J. Wawrzynek. Garp: A MIPs Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machine*s, page 12, 1997.

[HW99] S. Hauck and W. Wilson. Run-length Compression Techniques for FPGA Configurations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, page 286, 1999.

[JA04] Ahmed A. Jerraya. Long Term Trends for Embedded System Design. In *Proceedings of the 2004 Euromicro Symposium on Digital System Design*, pages 20-26, 2004.

[JC99] J. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 145-154, 1999.

[KH06] T. Kangas, T.D. Hämäläinen, and K. Kuusilinna. Scalable Architecture for SoC Video Encoders. In *the Journal of VLSI Signal Processing Systems*, volume 44, issue 1-2, pages 79-95, 2006.

[KR06] Ian Kuon and Jonathan Rose. Measuring and Closing the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th international Symposium on Field Programmable Gate Array*, page 21-30, 2006.

[Kra03] Jerry Krasner, Embedded Market Forecasters. *Total Cost of Development: A comprehensive cost estimation framework for evaluating embedded system platforms*, 2003.

[LV04] Roman Lysecky and Frank Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In *Proceedings of the 2004 Automation and Test in Europe*, volume 1, page 10480, 2004.

[LV05] Roman Lysecky, Frank Vahid. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *Proceedings of the* 2005 *Design, Automation and Test in Europe,* volume 1, pages 18-23, 2005.

[MD05] B. Miramond, J.M. Delosme. Decision guide environment for design space exploration. In *Proceedings of 10th IEEE Conference on Emerging Technologies and Factory Automation*, pages 811-817, 2005.

[Med06] MediaBench Consortium. Mediabench I Benchmarks. World Wide Web Document, 2006.
http://euler.slu.edu/~fritts/mediabench/

[Mip06] MIPS Technologies. MIPS Software Tools SDE Lite. World Wide Web Document, 2006.
http://www.mips.com/products/softwaretools/software_tools/MIPS_SDE_Lite.php

[Ope06] Open SystemC Initiative, SystemC Community, World Wide Web Document, 2006.
http://www.systemc.org/

[PD01] D. Panigrahi, S. Dey, R. Rao, K. Lahir, C. Chiasserini, and A. Raghunathan. Battery Life Estimation of Mobile Embedded Systems. In *Proceedings of the 14th international Conference on VLSI Design*, page 57, January 2001.

[PS03] Ryan Peoples and Todd Schuman. A Joint Performance and Financial Approach to Aircraft Design Optimization. Course Project Report, Massachusetts Institute of Technology, 2003.
http://ocw.mit.edu/OcwWeb/Aeronautics-and-Astronautics/16-888Spring 2004/Projects/index.htm

[PS99] Gianluca Palermo, Cristina Silvano and Vittorio Zaccaria. *A Flexible Framework for Fast Multi-objective Design Space Exploration of Embedded Systems*. Springer Verlag, Germany, Berlin, 1999.

[RJ03] Jeffry T. Russell and Margarida F. Jacome. Architecture-Level Performance Evaluation of Component-Based Embedded Systems. In *Proceedings of the 40th Design Automation Conference*, pages 396-401, 2003.

[Sau00] Gabriele Saucier. FPGA Technology Snapshot: Current Devices and Design Tools. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping*, page 200, 2000.

[SJ02] L. Shang and N. K. Jha. Hardware-Software Co-Synthesis of Low Power Real- Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs. In *Proceedings of the 2002 Conference on Asia South Pacific Design automation/VLSI Design*, page 345, 2002.

[SK02] L. Shang, A. S. Kaviani and K. Bathala. Dynamic power consumption in Virtex-II FPGA family. In *Proceedings of the 2002 ACM/SIGDA Tenth international Symposium on Field-Programmable Gate Arrays*, pages 157-164, 2002.

[SL03] Greg Stitt, Roman Lysecky and Frank Vahid. Dynamic Hardware/Software Partitioning: A First Approach. In *Proceedings of the 40th Design Automation Conference*, page 250, 2003.

[Smi97] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison Wesley, 1997.

[ST02] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson. A Dynamically Reconfigurable Adaptive Viterbi Decoder. In *Proceedings of the ACM/SIGDA International Symposium on FPGAs*, pages 227-236, 2002.

[Tan06] Adrian Tan. Being prepared for the Future of GNSS Positioning. In *Proceedings of the 2006 DigiTimes Tech Forum*, 2006.

[Tes04] Russel Tessier. Parallel Computer Architecture. Lecture Notes, 2004. http://www.ecs.umass.edu/ece/tessier/courses/669/index.html.

[Tes05] Russel Tessier. Reconfigurable Computing. Lecture Notes, 2005. http://ccnet.utoronto.ca/20059/ece1718hf/

[Thi06] Lothar Thiele. Hardware/Software Codesign. Lecture Notes, 2006. http://www.tik.ee.ethz.ch/tik/education/lectures/hswcd.

[VS02] J. Villarreal, D. Suresh, G. Stitt, F. Vahid and W. Najjar. Improving Software Performance with Configurable Logic. In *the Kluwer Journal on Design Automation of Embedded Systems*, volume 7, issue 4, pages 325-339, November 2002.

[VT02] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Inc., 2002.

[WW04] Olivier de Weck and Karen Willcox. Multidisciplinary System Design Optimization. Lecture Notes, 2004. http://ocw.mit.edu/OcwWeb/Aeronautics-and-Astronautics/16-888Spring-2004/CourseHome/index.htm

[Xil06a] Xilinx, Inc., San Jose, California. Virtex Series Family FPGAs. World Wide Web Document, 2006. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/

[Xil06b] Xilinx, Inc., San Jose, California. *Virtex-4 Configuration Guide*, January 2006

[Xil06c] Xilinx, Inc., San Jose, California. *Xilinx Virtex 4 family overview*, February 2006.

[YP05] P. Yiannacouras, J. Rose and J. G. Steffan. The Microarchitecture of FPGA Soft Processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 202-212, 2005.

Appendix A
Simulation Inputs

A.1 Application Models and Test Cases

The following software application models were used in the experiments to test the performance of the configuration scheduling algorithms. Each of the software application models has a number of operational modes, through which the application moves during its execution. It is assumed that the application stays in each operational mode for an equal period of time. The statistics describing the different models of the software application's behavior are presented in the following tables.

Table A-1: Temporal Locality Test Case 1

Mode	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
1	0%	100%	0%	0%	0%
2	0%	0%	100%	0%	0%
3	100%	0%	0%	0%	0%
4	0%	0%	0%	100%	0%
5	0%	0%	0%	0%	100%

Table A-2: Temporal Locality Test Case 2

Mode	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
1	2%	91%	3%	1%	3%
2	90%	2%	2%	3%	3%
3	3%	1%	92%	3%	1%
4	2%	2%	3%	91%	2%
5	4%	4%	1%	2%	89%

Table A-3: Temporal Locality Test Case 3

Mode	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
1	1%	2%	73%	4%	20%
2	5%	71%	18%	1%	5%
3	68%	20%	4%	6%	2%
4	3%	5%	1%	22%	69%
5	19%	1%	7%	70%	3%

Table A-4: Kernel Correlation Test Case 1

Previous Kernel	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
Mode 1					
Kernel 1	0%	100%	0%	0%	0%
Kernel 2	0%	0%	100%	0%	0%
Kernel 3	0%	0%	0%	0%	100%
Kernel 4	100%	0%	0%	0%	0%
Kernel 5	0%	0%	0%	100%	0%
Mode 2					
Kernel 1	0%	0%	100%	0%	0%
Kernel 2	0%	0%	0%	100%	0%
Kernel 3	0%	100%	0%	0%	0%
Kernel 4	0%	0%	0%	0%	100%
Kernel 5	100%	0%	0%	0%	0%
Mode 3					
Kernel 1	0%	0%	0%	100%	0%
Kernel 2	0%	0%	0%	0%	100%
Kernel 3	100%	0%	0%	0%	0%
Kernel 4	0%	100%	0%	0%	0%
Kernel 5	0%	0%	100%	0%	0%

Table A-5: Kernel Correlation Test Case 2

Previous Kernel	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
Mode 1					
Kernel 1	1%	89%	4%	6%	0%
Kernel 2	0%	3%	85%	7%	5%
Kernel 3	2%	3%	2%	5%	88%
Kernel 4	84%	0%	14%	0%	2%
Kernel 5	4%	3%	7%	83%	3%
Mode 2					
Kernel 1	0%	3%	85%	7%	5%
Kernel 2	4%	3%	7%	83%	3%
Kernel 3	1%	89%	4%	6%	0%
Kernel 4	2%	3%	2%	5%	88%
Kernel 5	84%	0%	14%	0%	2%
Mode 3					
Kernel 1	4%	3%	7%	83%	3%
Kernel 2	2%	3%	2%	5%	88%
Kernel 3	84%	0%	14%	0%	2%
Kernel 4	1%	89%	4%	6%	0%
Kernel 5	0%	3%	85%	7%	5%

Table A-6: Kernel Correlation Test Case 3

Previous Kernel	Probability of Kernel Executing next				
	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
Mode 1					
Kernel 1	2%	68%	10%	15%	5%
Kernel 2	11%	3%	71%	10%	5%
Kernel 3	12%	13%	7%	0%	68%
Kernel 4	69%	1%	14%	6%	10%
Kernel 5	3%	5%	10%	72%	10%
Mode 2					
Kernel 1	11%	3%	71%	10%	5%
Kernel 2	3%	5%	10%	72%	10%
Kernel 3	2%	68%	10%	15%	5%
Kernel 4	12%	13%	7%	0%	68%
Kernel 5	69%	1%	14%	6%	10%
Mode 3					
Kernel 1	3%	5%	10%	72%	10%
Kernel 2	12%	13%	7%	0%	68%
Kernel 3	69%	1%	14%	6%	10%
Kernel 4	2%	68%	10%	15%	5%
Kernel 5	11%	3%	71%	10%	5%

A.2 Timing Delays and Simulation Parameters

Some estimated values of timing delays and parameters, which were used in the study, are shown in Table A-7. The *history_buffer_length* is the length of the circular buffer, which indicates the maximum number of entries the buffer can hold.

Table A-7: Timing Delays and Parameter Values

Term	Value
N	5
t_{SW_1}	942.5 ms
t_{SW_2}	3595.9 ms
t_{SW_3}	2595.3 ms
t_{SW_4}	349.5 ms
t_{SW_5}	61.3 ms
t_{CHW_1}	483.0 ms
t_{CHW_2}	747.1 ms
t_{CHW_3}	154.1 ms
t_{CHW_4}	66.5 ms
t_{CHW_5}	0.7 ms
$t_{config\ CHW_n}$	26.7 ms for all n
$t_{start\ CHW_n}$	450 ns for all n
$t_{finish\ CHW_n}$	450 ns for all n
$t_{initiate}$	60 ns
t_{check}	60 ns
$t_{update\ TLCSA}$	135 ns
$t_{selection\ TLCSA}$	(1710 +135 * *history_buffer_length*) ns
$t_{update\ KCCSA}$	345 ns
$t_{selection\ KCCSA}$	(1785 +135 * *history_buffer_length*) ns

.

Appendix B

Simulation Results

The simulation results data for the various experiments are listed in the following tables.

B.1 Frequency Results

Table B-1: Temporal Locality Test Case 1 Frequency Results

Term	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
$P_{CHW_n \text{ not cfg}_{ODCSA}}$ =FRC	2.5%	2.5%	2.5%	2.5%	2.5%
$P_{CHW_n \text{ not cfg}_{TLCSA}}$	10%	10%	10%	10%	10%
$P_{CHW_n \text{ reconfig}_{TLCSA}}$ =FRC	2.5%	2.5%	2.5%	2.5%	2.5%
					history_buffer_length = 6 $Exec_{per\ mode}$ = 40

Table B-2: Temporal Locality Test Case 2 Frequency Results

Term	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
$P_{CHW_n \text{ not cfg}_{ODCSA}}$ =FRC	20.6%	18.5%	17.6%	19.6%	21.2%
$P_{CHW_n \text{ not cfg}_{TLCSA}}$	18.3%	17.1%	17.3%	18.5%	18.3%
$P_{CHW_n \text{ reconfig}_{TLCSA}}$ =FRC	2.5%	2.6%	2.5%	2.5%	2.6%
					history_buffer_length = 6 $Exec_{per\ mode}$ = 40

Table B-3: Temporal Locality Test Case 3 Frequency Results

Term	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
$P_{CHW_n \text{ not cfg}_{ODCSA}}$ =FRC	48.3%	45.1%	47.3%	48.9%	49.6%
$P_{CHW_n \text{ not cfg}_{TLCSA}}$	40.7%	37.2%	40.0%	41.0%	41.9%
$P_{CHW_n \text{ reconfig}_{TLCSA}}$ =FRC	5.77%	5.76%	5.77%	5.79%	6.27%
					history_buffer_length = 6 $Exec_{per\ mode}$ = 40

B.2 History Buffer Experiment Results

Table B-4: Temporal Locality Test Case 2 History Buffer Results

history_buffer_length	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Average
$P_{CHW_n \text{ not cfg}_{ODCSA}}$ = **FRC**						
-	20.6%	18.5%	17.6%	19.6%	21.2%	21.2%
$P_{CHW_n \text{ not cfg}_{TLCSA}}$						
2	17.5%	14.6%	14.3%	14.3%	17.4%	15.6%
4	17.3%	16.2%	15.9%	16.1%	16.3%	16.4%
6	18.3%	17.1%	17.3%	18.5%	18.3%	17.9%
8	20.7%	20.6%	19.0%	19.6%	19.3%	19.8%
10	24.0%	22.0%	22.0%	21.3%	21.9%	22.2%
15	28.8%	27.1%	26.7%	26.9%	27.8%	27.5%
20	35.4%	32.4%	32.6%	33.1%	33.1%	33.3%
$P_{CHW_n \text{ reconfig}_{TLCSA}}$ = **FRC**						
2	4.81%	3.87%	3.79%	3.50%	3.45%	3.88%
4	2.59%	2.76%	2.65%	2.69%	2.76%	2.69%
6	2.48%	2.63%	2.54%	2.53%	2.62%	2.56%
8	2.47%	2.48%	2.47%	2.51%	2.57%	2.50%
10	2.44%	2.52%	2.53%	2.50%	2.54%	2.51%
15	2.48%	2.48%	2.49%	2.49%	2.56%	2.50%
20	2.50%	2.51%	2.53%	2.46%	2.56%	2.51%
						$Exec_{per\ mode}$ = 40

Table B-5: Temporal Locality Test Case 3 History Buffer Results

history_buffer_length	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Average
$P_{CHW_n \text{ not cfg}_{ODCSA}}$ **=FRC**						
-	48.3%	45.1%	47.3%	48.9%	49.6%	47.8%
$P_{CHW_n \text{ not cfg}_{TLCSA}}$						
2	40.4%	35.8%	40.8%	42.5%	44.6%	40.8%
4	40.1%	36.8%	37.7%	41.5%	41.1%	39.5%
6	40.7%	37.2%	40.0%	41.0%	41.9%	40.1%
8	40.9%	36.0%	36.7%	40.5%	43.2%	39.5%
10	42.6%	36.1%	37.2%	40.6%	40.3%	39.4%
15	44.9%	40.4%	41.4%	42.9%	45.1%	43.0%
20	52.3%	44.0%	45.0%	46.2%	48.0%	47.1%
$P_{CHW_n \text{ reconfig}_{TLCSA}}$ **=FRC**						
2	13.75%	11.37%	11.09%	11.38%	10.77%	11.67%
4	8.50%	7.56%	7.08%	8.03%	8.49%	7.93%
6	5.77%	5.76%	5.77%	5.79%	6.27%	5.87%
8	4.22%	3.78%	3.50%	4.12%	4.78%	4.08%
10	3.82%	2.98%	2.96%	3.25%	3.28%	3.26%
15	2.92%	2.90%	2.55%	3.09%	3.01%	2.89%
20	2.84%	2.78%	2.50%	2.50%	2.70%	2.66%
$Exec_{per\ mode}$ = 40						

B.3 Execution Time Results

Table B-6: Temporal Locality Test Case 1 Execution Time Results

Term	SW_{system}	CHW_{stc_system}	CHW_{od_system}	CHW_{tl_system}
t_{total}	34,754 s	22,570 s	22,576 s	23,788 s
				history_buffer_length = 6
				$Exec_{per\ mode}$ = 40

Table B-7: Temporal Locality Test Case 2 Execution Time Results

Design Alternative	Reconfiguration Frequency and Time					
	1000 kHz	**100 kHz**	**50 kHz**	**10 kHz**	**8 kHz**	**5 kHz**
	27 ms	**267 ms**	**533 ms**	**2,667 ms**	**3,333 ms**	**5,333 ms**
SW_{system}	34,812 s					
CHW_{stc_system}	22,606 s					
CHW_{od_system}	22,658 s	23,126 s	23,645 s	27,803 s	29,102 s	33,000 s
CHW_{tl_system}	24,720 s	24,769 s	24,727 s	24,742 s	24,746 s	24,972 s
						history_buffer_length = 6
						$Exec_{per\ mode}$ = 40

Table B-8: Temporal Locality Test Case 3 Execution Time Results

Design Alternative	Reconfiguration Frequency and Time					
	1000 kHz	**100 kHz**	**50 kHz**	**10 kHz**	**8 kHz**	**5 kHz**
	27 ms	**267 ms**	**533 ms**	**2,667 ms**	**3,333 ms**	**5,333 ms**
SW_{system}	35,471 s					
CHW_{stc_system}	23,014 s					
CHW_{od_system}	23,142 s	24,289 s	25,564 s	35,763 s	38,950 s	48,511 s
CHW_{tl_system}	27,846 s	27,607 s	27,632 s	27,835 s	27,662 s	27,878 s
						history_buffer_length = 6
						$Exec_{per\ mode}$ = 40

Table B-9: Kernel Correlation Test Case 1 Execution Time Results

Design Alternative	Reconfiguration Frequency and Time					
	1000 kHz	**100 kHz**	**50 kHz**	**10 kHz**	**8 kHz**	**5 kHz**
	27 ms	**267 ms**	**533 ms**	**2,667 ms**	**3,333 ms**	**5,333 ms**
SW_{system}	34,760 s					
CHW_{stc_system}	22,573 s					
CHW_{od_system}	22,840 s	25,240 s	27,906 s	49,238 s	55,904 s	75,903 s
CHW_{kc_system}	23,575 s	23,586 s	23,758 s	30,912 s	32,092 s	33,747 s
history_buffer_length = 1 $Exec_{per\ mode}$ = 60						

Table B-10: Kernel Correlation Test Case 2 Execution Time Results

Design Alternative	Reconfiguration Frequency and Time					
	1000 kHz	**100 kHz**	**50 kHz**	**10 kHz**	**8 kHz**	**5 kHz**
	27 ms	**267 ms**	**533 ms**	**2,667 ms**	**3,333 ms**	**5,333 ms**
SW_{system}	34,976 s					
CHW_{stc_system}	22,680 s					
CHW_{od_system}	22,937 s	25,248 s	27,816 s	48,363 s	54,784 s	74,046 s
CHW_{kc_system}	26,557 s	26,613 s	26,788 s	31,381 s	32,244 s	33,624 s
history_buffer_length = 3 $Exec_{per\ mode}$ = 60						

Table B-11: Kernel Correlation Test Case 3 Execution Time Results

Design Alternative	Reconfiguration Frequency and Time					
	1000 kHz	**100 kHz**	**50 kHz**	**10 kHz**	**8 kHz**	**5 kHz**
	27 ms	**267 ms**	**533 ms**	**2,667 ms**	**3,333 ms**	**5,333 ms**
SW_{system}	34,484 s					
CHW_{stc_system}	22,377 s					
CHW_{od_system}	22,623 s	24,832 s	27,287 s	46,924 s	53,061 s	71,472 s
CHW_{kc_system}	28,235 s	28,277 s	28,384 s	31,377 s	31,982 s	32,887 s
history_buffer_length = 3 $Exec_{per\ mode}$ = 60						

Appendix C

Glossary of Acronyms

ASIC – Application Specific Integrated Circuit
CHW – Configurable Hardware
CPI – Clock Cycles per Instruction
CPU – Central Processing Unit
CSA – Configuration Scheduling Algorithm
DHSP – Dynamic Hardware/Software Partitioning
DHW – Dedicated Hardware
EDA – Electronic Design Automation
FAC – Frequency of Already Configured
FPGA – Field Programmable Gate Array
FRC – Frequency of Reconfiguration
GPP – General Purpose Processor
HDL – Hardware Description Language
IC – Integrated Circuit
KC – Kernel Correlation
OD – On Demand
PC – Personal Computer
RTL – Register Transfer Level
SPP – Single Purpose Processor
TL – Temporal Locality
TTM – Time to Market